UIUCDCS-R-80-1027

UILU-ENG 80 1726

HARDWARE FOR SEARCHING VERY LARGE TEXT DATABASES

by

Roger Lee Haskin

August 1980

DEPARTMENT OF COMPUTER SCIENCE
UNIVERSITY OF ILLINOIS AT URBANA-CHAMPAIGN · URBANA, ILLINOIS

HARDWARE FOR SEARCHING VERY LARGE TEXT DATABASES

BY

ROGER LEE HASKIN

B.S., University of Illinois at Urbana-Champaign, 1973
M.S., University of Illinois at Urbana-Champaign, 1978

THESIS

Submitted in partial fulfillment of the requirements
for the degree of Doctor of Philosophy in Computer Science
in the Graduate College of the
University of Illinois at Urbana-Champaign, 1980

Urbana, Illinois

HARDWARE FOR SEARCHING VERY LARGE TEXT DATABASES

Roger Lee Haskin, Ph.D.
Department of Computer Science
University of Illinois at Urbana-Champaign, 1980

The problems involved in searching very large text databases are discussed. It is shown that conventional techniques for searching current databases cannot be scaled up to larger ones, and that it is necessary to build hardware to search the database in parallel if reasonable search times are expected. The part of the search process requiring the highest bandwidth is scanning the database to detect instances of search terms. Methods from the literature of doing this in hardware are examined, problems with using them in large systems are discussed, and design criteria to be met by a successful search architecture are defined.

A new model for text searching, using a nondeterministic finite state automaton (NFSA) to control matching, is introduced. First the NFSA model itself is discussed. Examples are given showing how it can be used to search for a wide variety of textual patterns. Next, implementation of the NFSA Searcher in hardware is discussed. By partitioning the nondeterministic state table on the basis of pairwise compatibilities and assigning blocks of states to interconnected sub-machines, it is shown how the NFSA can be built with simple logic in a manner that lends itself to LSI implementation.

It is of critical importance that it be possible to quickly partition the state table for a group of search patterns. Methods for partitioning tables efficiently are developed and their performance is analyzed. Methods of detecting instances of higher-level search expressions from instances of their component patterns detected by the NFSA Searcher are also discussed. Finally, the configuration and performance of the search system as a function of user load and other paramenters is discussed. By comparing the hardware required and response time afforded by the NFSA Searcher with that for an alternative implementation, it is seen that the NFSA Searcher is a significant advance in architecture for text searching.

Table of Contents

Chapter 1

Introduction

Several large scale online text retrieval systems are now available, both systems that store document abstracts (Medline, [McCar78], Orbit [Black78], and Dialog [Bayer78] to name a few), and those that store the full text of their documents (notably Lexis [Sprowl76], and Westlaw and Juris [Hollaar79]). Large amounts of computer readable text are also being accumulated as a side effect of other processes such as office automation and computer typesetting. The availability of large text databases containing material pertinent to fields such as law, engineering, management, etc., will generate a demand for facilities to allow this data to be searched. Thus both the number and size of text database systems can be expected to increase in the near future.

Two factors have contributed to limiting the growth of such systems, both in terms of the database size, and in terms of the size of the user community that can afford to use them. These factors are the processing capacity of the computers on which the systems run, and the cost of online storage - both the per-bit costs of disk drives and the indirect cost of the space to put them. Disk technology is advancing rapidly; current drives have over 30 times the capacity and cost less than half as much as those of ten years ago. Databases in the 30-100 billion character size range are now economically feasible. New advances such as monolithic read-write heads and `Winchester′ (sealed surface) technology promise to continue this trend.

Mainframe technology has not kept pace - most of the speedup in recent computers has been in execution of floating-point operations. Systems whose query response times are only just tolerable to begin with cannot be expanded to take advantage of the increased database size allowed by larger, cheaper disks unless methods are found to speed up searching. However, this speedup must not have the effect of pricing retrieval systems out of the reach of of their user community, as just

adding more conventional processing power would do. Searches on present systems are too expensive already (from a few dollars to as much as $100 per search); any significant increase in this cost would be unacceptable.

Much effort has been directed toward developing special purpose hardware to speed searching of formatted databases. In particular, machines to support the relational model (RARES [LiSmSm76], RAP [OzScSm75], and CASSM [SuLip75]), and variants of this model (i.e. sets of value-attribute tuples, for example DBC [HsKaKe75]) have been proposed. Prototypes of RAP and CASSM have been built. Unfortunately, text (journal articles, legal decisions, books, etc.) cannot usually be formatted into the fixed size fields normally required by such database management systems. Thus, these hardware designs are not directly applicable for use as fast text searchers.

1.1 Search Strategies

Strategies for speeding up the search of text databases are either to increase the speed at which characters in the database can be processed, or to narrow down the area to be searched to a small fraction of the database. The second strategy usually involves using an index that contains pointers to occurrences of search terms in the data. Index-processing hardware has been relatively well studied ([Stell75], [Hollaar76], [Hurl76], and [Miln76]).

It is tempting to place the entire processing burden on the index processor by inverting to the word level. Here, no text searching at all is necessary. While it initially appears to be a viable solution, word-level inversion has severe drawbacks. One is the complexity of the postings (pointers to term occurrences). The index processors proposed in the literature are only designed to perform simple AND, OR, and NOT operations between postings lists, and this is not adequate to handle common query operations such as proximity (`A within five words of B') and context qualification (`A and B in the same sentence'). Another problem is the large amount of space required to store a fully inverted index. Space estimates for storing the index range from 50% to 300% of the space needed to store the text itself([BiNeTr78]). The performance of index

processors such as Stellhorn's degrades substantially if the postings lists are stored on moving head disks ([Hurl76],[Miln76]). Storing an index of a size comparable to the full text is expensive enough on moving head disks, and is not economically feasible on any faster medium (i.e. drum, shift register memory, etc.). Thus, the performance degradation due to the use of moving head disks for index storage must be accepted if full inversion is used.

Similarly, searching the entire database for each query has been proposed [BiNeTr78]. In this strategy, the entire surface of each disk is scanned repeatedly. During each scan, the batch of queries arriving during the preceding scan is searched for. Searching the entire database results in slow best case response time. How slow response is depends upon the implementation (whether all drives are searched in parallel or sequentially; whether multiple heads per drive are scanned simultaneously, etc.). This topic will be discussed in more detail in Chapter 5.

A third alternative search strategy is partial inversion. Here, the index contains a posting for each region of the database (i.e. cylinder, track, document, etc.) containing an instance of the term. This index contains fewer and smaller postings, so the index size can be held to around 20% to 30% of the text size. It thus costs much less to store than the considerably larger fully inverted index. The index is processed as for a fully inverted system, and then regions in the result postings list are searched for actual occurrences of the search expression. Since the region pointed to by each posting can (and usually does) contain several occurrences of the term, postings lists will be much shorter and faster to process than if full inversion were used. By using a partially inverted index, it might be possible to confine full text searching to a small enough area to make it reasonable to use a conventional processor for searching. However, even assuming that such a processor could keep up with disk transfer rates (about 1 microsecond per character), and assuming that the index processor will narrow the search to a very optimistic 0.1% of a 50-billion character database, the average query would completely saturate the processor for almost a minute. A system of this size would be expensive and require many users to pay for its operation. If a

reasonable number of users are to be accommodated simultaneously, it is clearly quite intolerable to require a full minute for processing an individual query!

No matter whether full searching or partial inversion is used, special search hardware is necessary if the database is large and if fast response time is desired. Thus, a general-purpose large text database system might consist of a conventional computer to handle communication with users and to translate queries, of index and text data residing in secondary storage, and of special purpose machines to process the index and search the text.

1.2 Text Searchers

Before discussing the design of a text searcher, its function should be more clearly defined. The searcher (Figure 1.1) can be thought of as a black box accepting questions (search commands) consisting of an encoded query (search expression) and the address of a region to be searched. It responds with pointers to occurrences of the search expression in the region. The searcher accepts many such commands, processing them concurrently. It does not necessarily provide answers in the order the questions were issued, but rather responds in an order that attempts to optimize some performance variable such as throughput or response time.

The search system thus has the job of scheduling the order in which regions are to be searched, and of carrying out the searches. Searching itself can be broken down into two processes: scanning the database region being searched for instances of search terms and formatting codes (i.e. end of sentence, etc.), and doing the bookkeeping necessary to detect instances of the query's search expression (query resolution).

1.3 Query Resolution

Conceptually, query resolution is quite simple. Each time a term is matched, the occurrence is recorded, and at each context boundary the occurrences are checked to see whether the search expression has been satisfied. Then the data structure is reinitialized (the appropriate

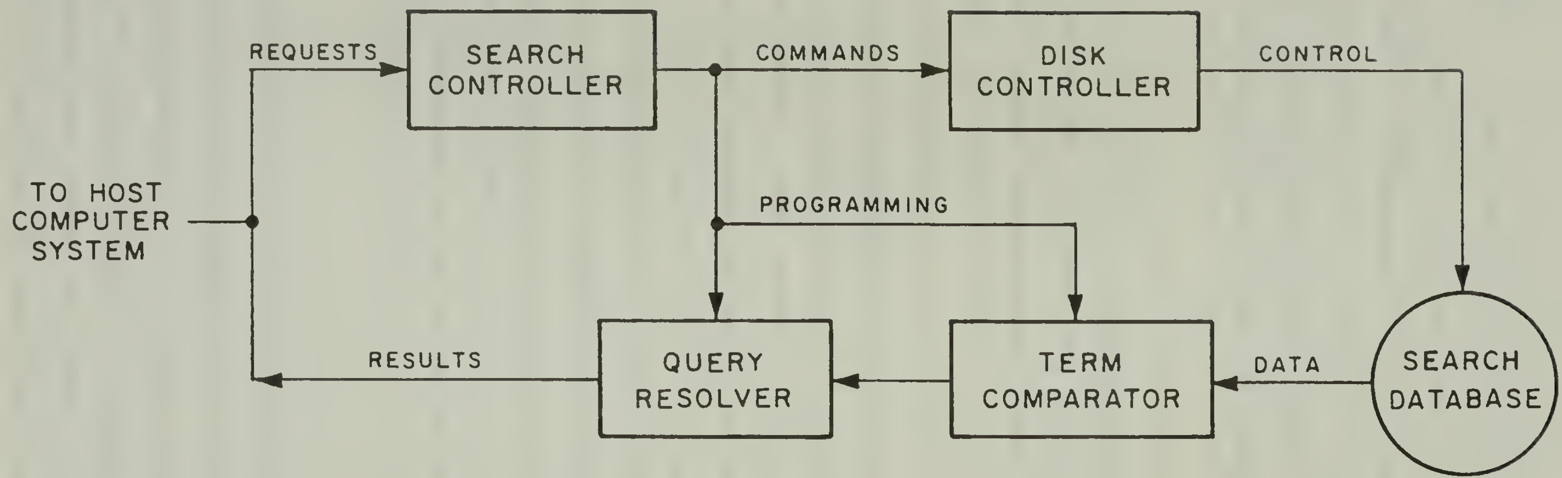


Figure 1.1 Hardware Text Searcher

term-found flags are reset) and the search continues. Each time an instance of the expression is detected, its location is reported to the host.

Resolvers for conventional systems operate sequentially; documents are processed one at a time from start to finish. Design problems are typically dependent upon the implementation of the term matching logic and upon the search expression format - for example, matching resolution speed to expected term hit rates. Such considerations are often very mundane, and as might be expected, not much has appeared on query resolution in the literature.

The query resolution functions for the CIA's proposed SAFE system were discussed in [OSI77a] and [OSI77b]. Roberts [Rob77] discussed query resolution processing speed requirements under assumptions on query statistics appropriate to the SAFE system. The query resolver for this system is designed to operate in conjunction with a hardware term matcher, and is intended to perform query resolution at disk speeds. However, the SAFE system has several peculiarities which limit the applicability of its techniques to more general systems. The system has a relatively slow response time (nominally over seven minutes) and is used essentially in a batch mode. Thus, queries tend to be more complex than typical queries for other systems. Also, the SAFE system is not indexed. Since non-relevant as well as relevant documents are searched, the hit rate on each term can be expected to be somewhat lower than if only potentially relevant documents were searched. Finally, the query language limits the type of search operations which can be performed. Further development of the resolution algorithms are necessary if a flexible query language is to be supported.

Stellhorn ([Stell74a],[Stell74b]) discusses search expression data structures and resolution processing techniques used by the EUREKA system ([Morgan76],[BurEm79]). Most of this discussion assumes a fixed, hierarchically related set of formatting codes and a term matching technique that requires making one pass over the data for each term in the expression. The result was a set of heuristics for optimizing search time

under these constraints, but the work is of little relevance to systems capable of searching for many terms simultaneously.

Questions of speed, memory requirements, and efficient algorithms necessary in the parallel search system being considered here remain to be answered.

1.4 Term Matching - Previous Efforts

Several approaches to building term-matching hardware have been suggested in the literature. Cheng [Cheng77] proposed two parallel matcher designs using associative match cells (character comparators) to match strings held in memory at extremely high speeds. It was intended primarily to speed execution of SNOBOL pattern matches and has several drawbacks that make it impractical for use in searching databases. First, it was designed to search short (less than 100 character) strings - terms spanning string boundaries cannot be matched and must be processed specially. Second, a huge number of comparators is needed. The fastest version of the matcher uses an MxN array of comparators, where M is the string length (about 100 characters) and N is the maximum number of characters in a term (about 16). Either the matcher must be cycled once for each term being searched, or more than one NxM comparator array must be included. Large systems needing search hardware can require searching for many terms (50-100) simultaneously, making either of these alternatives unpalatable. Finally, duplicating this huge matrix for many disk drives is impractical, so the searcher must be built as a centralized unit with a large buffer memory, with the disks transferring into it in parallel with searching (Figure 1.2). The cabling and channel hardware to do this would be prohibitively costly.

Cabling, channel hardware, and buffer memory costs can be eliminated by placing the search hardware local to the storage device (Figure 1.3). In this arrangement, the only high-bandwidth path is between the storage medium and the term matcher connected to it. The only necessary communication between searchers and host is instructions telling the searchers what to look for, and reports to the host on the locations of hits on the search expression. The low bandwidth required by this

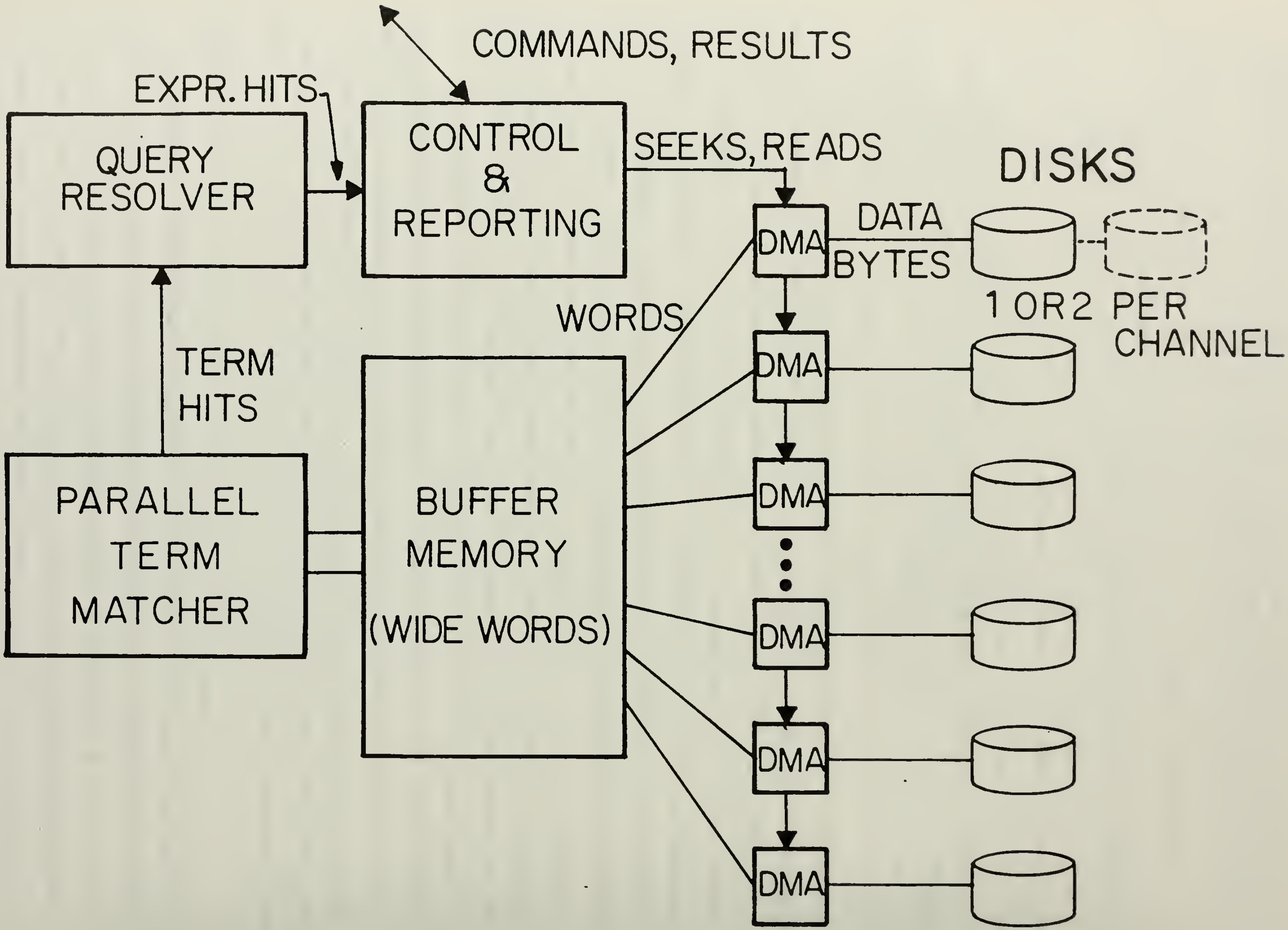


Figure 1.2 Centralized Searcher

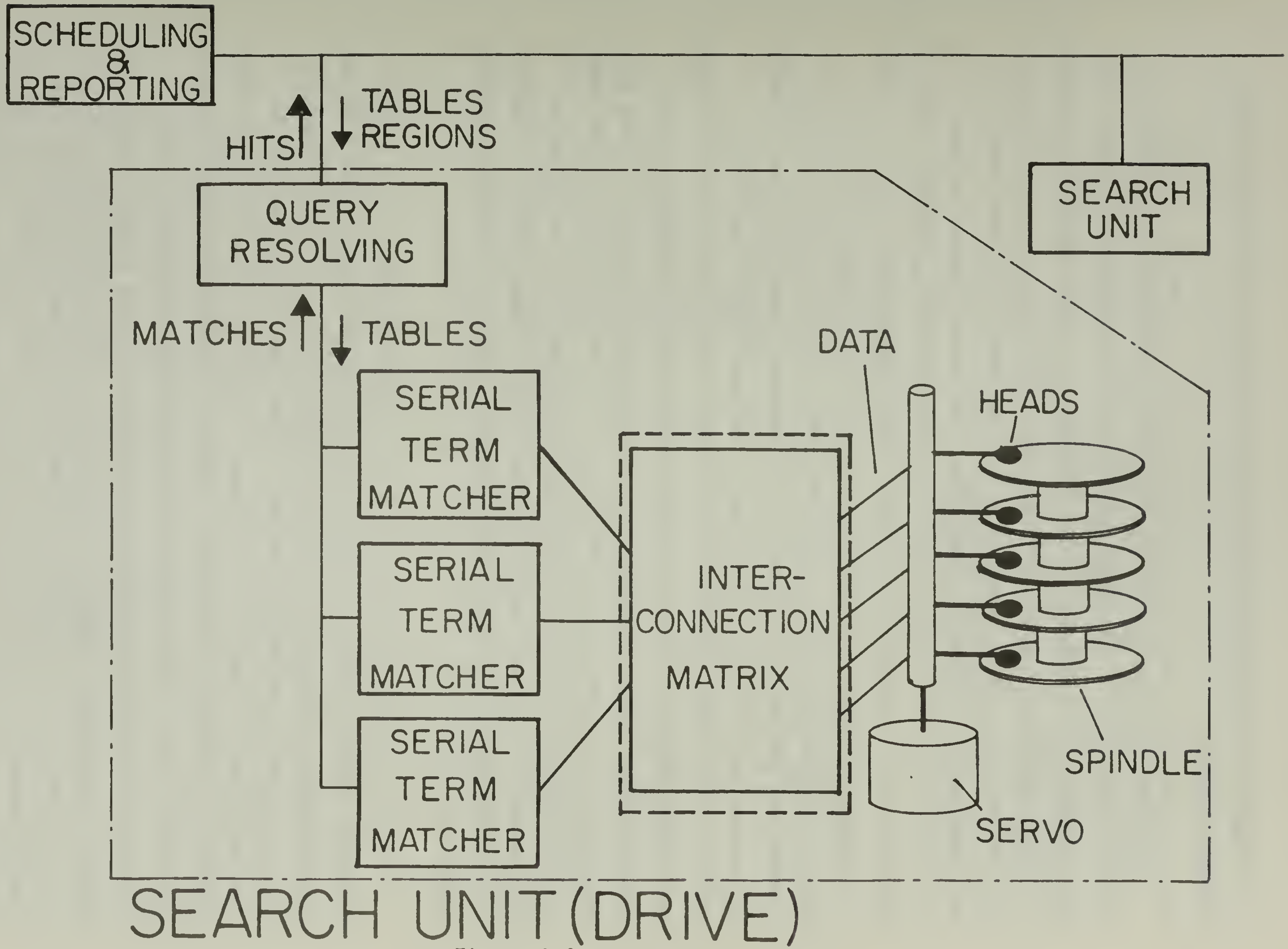


Figure 1.3 Distributed Searcher

communication path allows searchers to be connected to the host along a standard I/O bus.

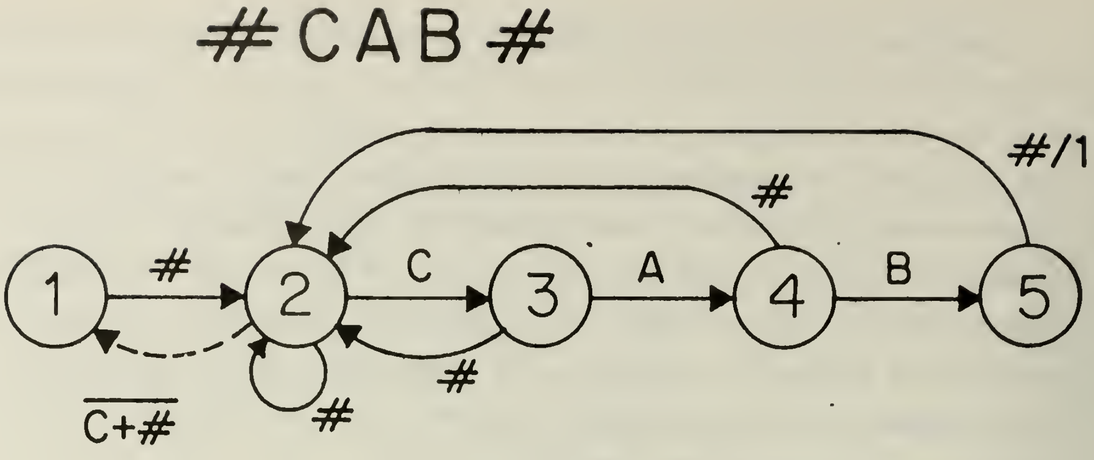
Distributed searchers using associative comparators to match terms have been proposed by Stellhorn [Stell74b], Bird [BiTuWo77], Foster and Kung [FoKu80], and Mules and Warter [MuWar79]. These matchers accept data at typical disk rates (200 - 1000 ns. per character), thus one matcher is required per track being searched at any one time. The first three of these matchers share similar drawbacks. They cannot handle embedded variable-length don't cares (EVLDC's - patterns with a specified prefix and suffix and an unspecified middle). More importantly, they are limited as to both the number and length of terms they can handle. Stellhorn's matcher, for example, uses a fixed NxM array of cells, where N is the maximum term length allowed and M is the maximum number of terms allowed. The average term length in most queries is under eight characters; allowing a reasonable maximum term length of 16 characters results in many cells being unused. Also, associative matchers are typically not very flexible in the types of matching they can perform (usually allowing only exact match or fixed-length don't care). It is very useful to have other functions available (such as matching any punctuation or matching any alphanumeric). However, adding any more power to the comparator requires that the necessary added logic be included in each cell in the comparator array.

Mules and Warter's matcher is a very clever design - it is capable of detecting several types of errors in the data (insertions, deletions, and transpositions) and matching terms in spite of them. It has a flexible bit-by-bit don't care masking scheme. This matcher was designed for searching a small, noninverted database. It is debatable how useful the error-detection feature is with an inverted database; the index processor will only point the text searcher to areas known to contain correct spellings. Also, this searcher is designed to handle a small number of terms (16) of limited length (16 characters). It is not clear that it could be economically scaled up to handle the number of terms required in a large system.

Copeland [Cop78] and Mukhopahhyay [Muk78] independently proposed matchers based upon networks of match cells. Copeland's matcher was used in a relational database environment, and matched only one term per pass over the data. Mukhopadhyay described a more complex design with both character match cells and other cells that handle functions such as character counting and boolean operations. Presumably a reconfigurable interconnection network would be required. This was not discussed in [Muk78], and since such networks are difficult to implement in LSI, a large matcher of this type would probably not be practical for direct scanning of data coming off the disk.

An alternative implementation for the term matcher can be based upon the concept of a finite-state automaton (FSA). Figure 1.4 shows the state diagram and associated table necessary to match the word `#CAB#', where `#' indicates a word break. When a `#' is seen, the FSA makes a transition to state 2. If the next input character is `C', the transition to state 3 is taken, and any other input causes the FSA to go either to state 2 (on a word break) or state 1 (for anything else). Matching continues in this manner until the trailing `#' is seen, at which point an output signal is produced and the FSA returns to state 2. Conventially, this state diagram is specified in the tabular form shown in Figure 1.4. Each row represents a state, and the entry in each column corresponds to the next state if the input character shown at the top of the column is received.

Bird [OSI77a] designed a searcher based upon an FSA model. This matcher (Figure 1.5) is designed to search the entire disk sequentially; all queries arriving during a search are batched and processed concurrently during the next pass. The number of terms in a batch of queries can be quite large. All terms to be searched for are collected, and a state table is built. A conventional FSA state table, with one row per state (character in a term) and one column per possible input character code, would require a prohibitive amount of memory to store. Recognizing that the table is very sparse, Bird devised a method of storing it in much less memory, using one type of state (sequential) for states having one transition leading out of them, and another type (index)



STATE	INPUT #	A	B	C	D	....
1	2	1	1	1	1	...
2	2	1	1	3	1	...
3	2	4	1	1	1	...
4	2	1	5	1	1	...
5	2/1	1	1	1	1	....

Figure 1.4 FSA State Diagram and Table

for the 10% of the states in the table having more than one outbound transition defined for them.

Sequential states operate straightforwardly; each has only one possible successful match character. If the next input character corresponds to the one required for the current state, a transition to the next sequential state is taken. On a mismatch, a <u>default</u> transition to an idle state is taken. States having more than one successful match alternative require a more sophisticated processing method, which is embodied in the <u>index</u> state. Rather than indexing into a vector of next

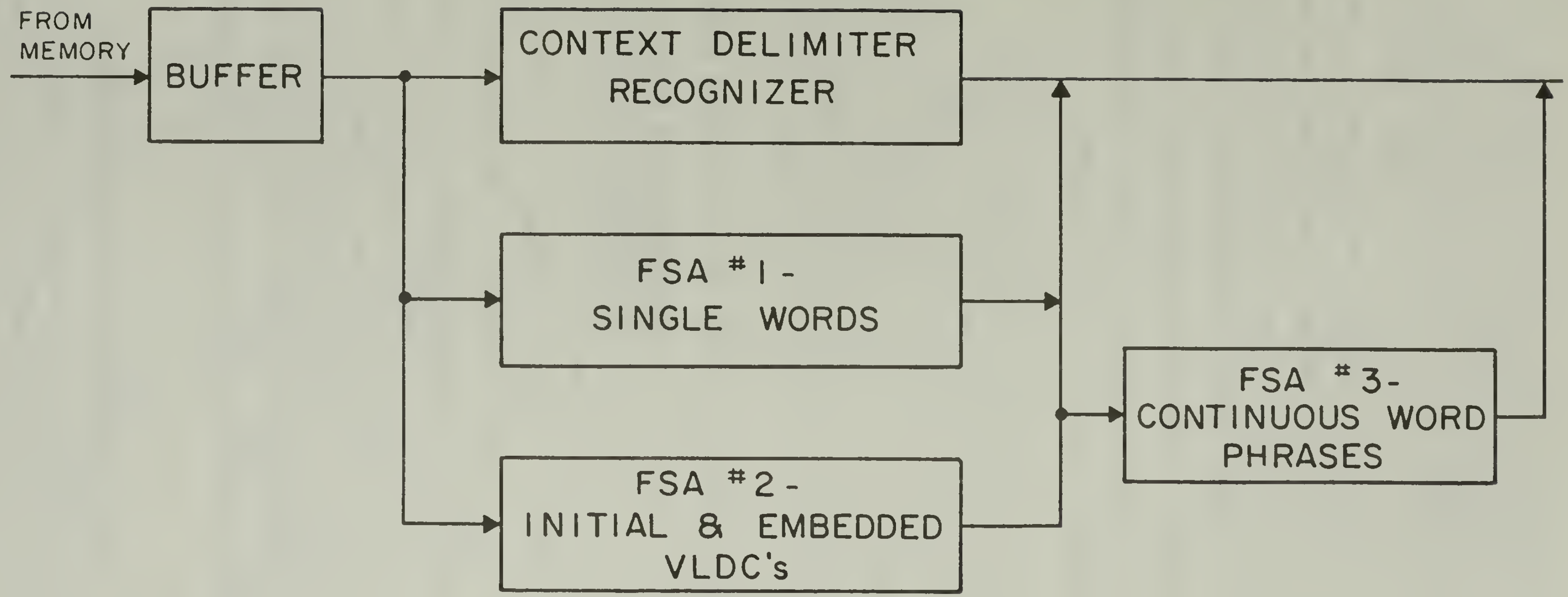


Figure 1.5 Bird FSA Term Matcher

state addresses, one for each possible input character (wasteful of memory) or comparing against a list of alternative successful match characters (requiring multiple comparators or iteratively cycling one comparator), the Bird machine has a bit vector stored in each index state word. Each character code in the input alphabet corresponds to a bit position in the vector. When an input character is received whose bit is set, a `leading ones' count is done. The number of bits set in the vector with positions lower than that of the input character's bit is used as an index, and added to a base address stored in the state word. The resultant address is that of the next state. If the input character's bit was clear, the default transition to an idle state is taken.

The FSA matcher represents a significant improvement over the previous designs. It has less memory wasted due to term-length variations, and since separate comparison logic is not required for each character in the list of terms, it can be fairly sophisticated in terms of what types of matching it will perform. Also, no complex interconnection network is required. However, the FSA has a few shortcomings of its own.

There is a definite speed/hardware tradeoff in the leading ones counter. Consider a Bird machine matching a database with a character set with n codes (typically, n is in the range from 64 to 256). Counting the ones in an n-bit vector can be done with one shifter and a log n bit counter in n steps, or can be done in O(log n * loglog n) time using O(n $\log^2$n loglog n) gates ([ChKu77]). If n is large or if the data rate is fast, the expensive realization may be necessary (repeated, of course, for every disk drive in the system). To overcome this, a version of the Bird machine was proposed which processes the data stream in 4-bit nibbles. This reduces the bit vector size to 16 bits and simplifies the leading-ones counter, but requires two states per character. As a result, the rate at which characters can be matched using a given state table memory speed is reduced.

The positional bit vector also complicates matching on character classes. If a particular transition was to be taken if the next input character was any numeric digit, all ten corresponding bits would have to

be set and ten separate (identical) states would be required in the list of next states. The Bird machine as implemented has special tests to overcome this problem, at the expense of extra hardware and complexity.

Certain terms (and combinations of terms) greatly increase the complexity of the FSA state table. This can be illustrated most simply with two examples. Figure 1.6 shows a state diagram to match the string `ANAS′. Suppose the input string is `BANANAS′. When the first `ANA′ has been matched, the FSA will be in state 3. When the following `N′ is seen, the FSA must `backtrack′ to state 2 to enable successful recognition of the succeeding `AS′ rather than take the normal mismatch transition to state 0. Similarly, if an `A′ is seen in any state, the FSA must go to state 1. Figure 1.7 shows the state diagram to match the two strings `FISH′ and `ISMS′. It illustrates that the start of a term can be encountered while another one is being matched. Terms such as these which are not required to begin on word boundaries (i.e. whose first character can be preceded by another alphanumeric character) are called initial variable-length don′t-cares (IVLDC′s). Notice first that each state must have transitions to state 1 (if `F′ is seen) and 4 (if `I′ is seen).

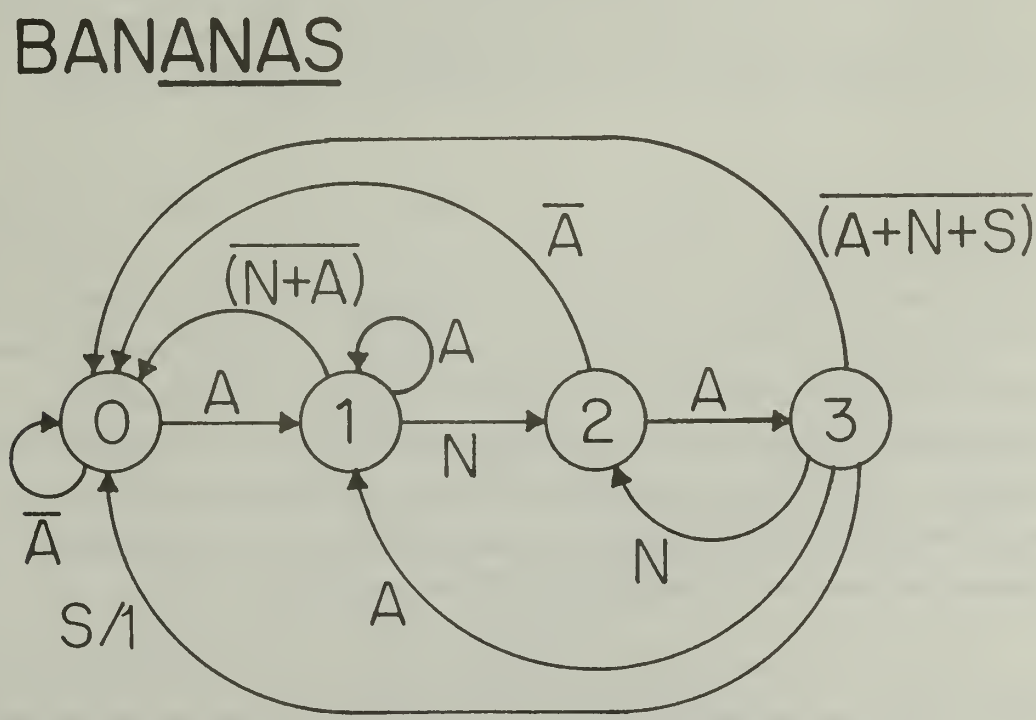


Figure 1.6 Diagram of FSA State Table with Backup Transitions

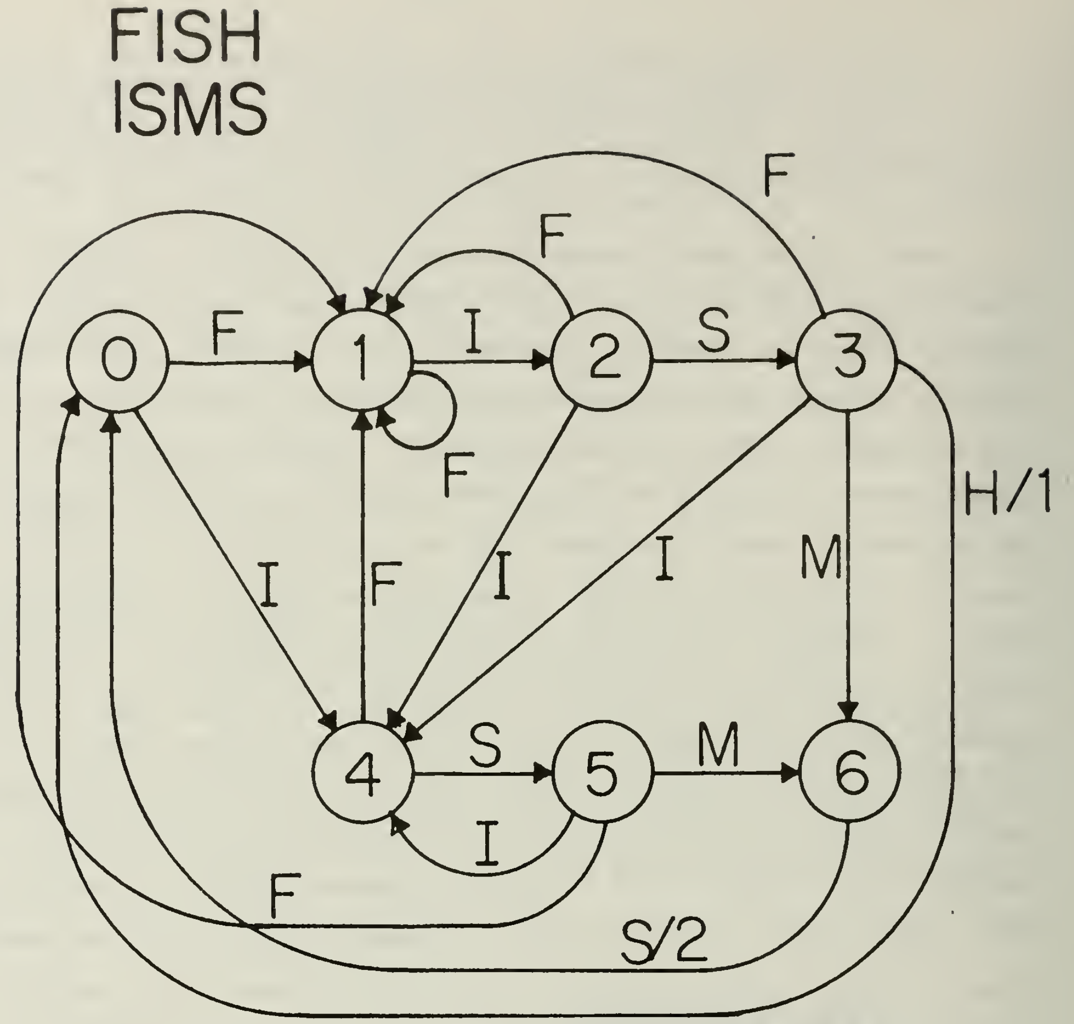


Figure 1.7 Diagram of FSA Table with Embedded Startups

Also, state 3 must 'remember' that 'ISMS' can occur within a suspected instance of 'FISH', and must have a transition to state 6 to allow this path to be continued. In Figure 1.7, mismatch transitions back to state 0 have been left out for simplicity, but they still exist in the table. The point of these examples is that every character of the input is potentially the start of a search term in addition to being the middle of another. Terms which can start during the matching of others (or themselves) must be detected when the state table is being built, and the appropriate recovery transitions must be included.

The Bird machine ameliorates this problem to some extent by including special logic to automatically execute transitions to one state if the

input is both a mismatch and a word separator, and another state on any other mismatch. If all terms start on word boundaries, most have only one match transition and can be assigned as sequential states. However, if even one term is not restricted to starting on a word boundary, all other states in the table must have recovery transitions leading to that term, and thus must be index states. To circumvent this problem, a second identical FSA is included in the Bird machine, and all IVLDC terms are assigned to it. All states in the second FSA are index states, but the main FSA can now be assigned mostly to sequential states. A similar problem exists with continuous word phrases (CWP's) - terms containing more than one word in sequence. These are handled by yet a third FSA.

The implementation of index states causes three more problems. First, sequential and index states require different amounts of time to process. The Bird machine uses a FIFO input buffer to synchronize the FSA with the disk. Second, the Bird machine requires memory access times in the 100 nsec. range. The per-bit cost of memory in this speed range is higher than that for slower memory. Finally, index states require storing the leading ones bit vector, the index table base address, and several other smaller fields. As a result the state memory word is quite wide (85 bits for the 6-bit character version and 39 bits for the 4-bit nibble version). The necessary high memory speed and wide data path both would complicate an LSI implementation.

1.5 Summary

This chapter discussed the problem of searching very large text databases. The applicability of indexing and full-text searching were discussed, and it was shown that a combination of the two techniques, partial inversion, was an attractive approach to use in implementing a search system. However, even if partial inversion is used to narrow the search to a fraction of the database, the necessary searcher bandwidth is too high to be accommodated by conventional processors, and specialized text search processors need to be developed.

There has been some investigation into the design of such processors, and how these designs could be used in a large-system environment was

discussed. The three basic organizations which have been explored in the literature are associative array comparators, comparator networks, and finite state automata. It was shown that each has inherent problems complicating their use in large systems. It is now possible to state several criteria to be met by a well-designed text searcher for large text databases:

1. The design should be flexible in accomodating different system loads. It should be configurable, such that different systems can be designed to handle different ranges of user loads and expected query complexities using the same basic design. The design should scale up well as the load it is to be configured to handle increases.
2. To minimize cost in large systems requiring many searchers, the design should lend itself to LSI implementation. Factors such as data path width, use of memory vs. random logic, and partitioning into small, identical building blocks should be considered.
3. It should straightforwardly implement complex matching operations such as matching groups of character codes, handling variable-length don't-cares (VLDC's), and recognizing document formatting codes.
4. The channel bandwidth should be minimized to lower cabling and hardware costs.
5. The design should not require the use of large buffer memories. Not only are these expensive, but extra program logic and execution time is required to allocate space in them.

Each of the three organizations discussed fell short in one or more of these areas. To facilitate building search systems for large databases, improved architectures are clearly needed. The subsequent chapters will be devoted to developing such an architecture. First, a new model for the search process will be introduced. Next, it will be shown how the model can be mapped into hardware in a manner fulfilling each of the above design criteria. Then, it will be shown how this searcher can be fitted into a retrieval system. Problems of translating queries into commands for the search system and of processing the output of the

searcher to detect instances of search expressions will be discussed, both in single-user and multi-user environments. Configuring the search hardware for use in individual systems will be discussed. It will be considered how parameters such as user load, index system selectivity, and query size affect both the configuration of the system (i.e. how much hardware is necessary to handle the load) and the response time. Finally, the new design will be compared with one of the above architectures to determine the cost and performance improvement it affords.

In short, this chapter demonstrated why it is difficult to build search systems for very large text databases using current designs. The following chapters will introduce a new design, and will show how large systems having excellent response times can be built using it.

Chapter 2

The NFSA Term Matcher

Of the alternatives examined so far, the Bird FSA came closest to fulfilling the design criteria stated at the end of Chapter 1. It was more economical to implement in hardware than the associative schemes, and could handle don't-care conditions without the difficulties of other approaches. The problems to be overcome were the necessary recovery (backtrack) transitions if wrong paths are taken, the many alternative transitions out of certain states, the wide state word necessitated by the leading-ones bit vector, and the fact that the state table must be stored in relatively fast memory. These problems are not due to the implementation of the FSA matcher; they are due to inherent shortcomings of the deterministic FSA as a model for the search process.

2.1 The NFSA Model

A far superior model for the text search process is the nondeterministic finite state automaton. An NFSA can be described ([HopU179]) as a 6-tuple $(\{X\},\{Y\},\{Z\},M,O,y_0)$:

- $\{X\}$ is the alphabet of allowable input characters
- $\{Y\}$ is the set of states the NFSA can occupy
- $\{Z\}$ is the set of outputs produced
- M is a mapping $X \times Y \rightarrow 2^Y$ (the state transition function. The set 2^Y is the power set of $\{Y\}$.)
- O is a mapping $X \times Y \rightarrow \{Z\}$ (the output function)
- y_0 is the initial state

The primary difference between a deterministic and nondeterministic FSA is that the latter can occupy more than one state at once. Actually, different references state this concept in different ways. For example, Hopcroft and Ullman ([HopU169]) define a NFSA as `choosing any one' of the successors of a state when making a transition. A sequence of inputs is

accepted if there exists a sequence of transitions leading into a final state that could be taken as a result of receiving the input sequence. (A final state would correspond to one generating an output in the above definition.) The ambiguity regarding which transition is taken at each step is presumably resolved upon arrival in the final state. Ten years later ([HopUl79]) they changed their minds; now during each transition with multiple successors, the NFSA is said to `make a duplicate copy of itself′ in each possible successor state. For our purposes these distinctions are meaningless, it is just as easy to think of the NFSA as one machine that can occupy several states simultaneously.

The FSA matched input strings by occupying one state and making a transition to one of a number of states based upon the input character. In any state the FSA had to determine which alternative input was received and choose one of a number of possible successors. The comparator logic to do this was quite complex. The NFSA will be employed to match terms by having it occupy one state for each alternative next input character. Instead of choosing from a number of successors, each state will only have to determine whether or not the next input is the one alternative that it is assigned to match, and make transitions to one (or more) successors if a match occurs. It is quite simple to design such a yes/no comparator.

To allow the use of a simple yes/no comparator, the transitions out of all states but the initial (or idle) state will be restricted in the following way: Each non-idle state will have associated with it some subset x of the input alphabet {X}, and all transitions out of the state will either be labelled x or $\overline{x}$ (the complement of subset x). Deciding which transition(s) to take out of a state only requires one check to see whether the input is in x. The form of these subsets x will be further restricted to include only certain useful combinations of the inputs (i.e. single characters, all alphanumerics, all punctuation, document formatting codes, etc.) which will allow building a comparator to evaluate an input′s membership in x using a minimum of logic.

Figure 2.1 shows a sample list of terms to be matched. The special character `#′ signals that any punctuation (word separator) can appear in

1. #A?ISM#
2. IST#
3. #SCHISM#
4. #BEST
5. #BENT#
6. #BUNT
7. #BUNTED#

Figure 2.1 Sample List of Search Terms

that position, `%' stands for any document formatting code (i.e. end of sentence), `*' denotes that any single character can appear, `α' allows any alphanumeric character, and `?' stands for any <u>string</u> of alphanumeric characters. Figure 2.2 shows the diagram of a nondeterministic state table generated directly from the term list. Each transition is labeled with the character that must be found for the transition to be taken. Mismatches cause transitions to be taken to the idle state, which is not shown for clarity. Also, transitions not leading into or out of numbered states are presumed to lead from or to the idle state.

The last transition for each term is labeled `x/z' where $x \varepsilon \{X\}$ is the last character in the term, and $z \varepsilon \{Z\}$ is the output produced that signals detection of the end of a term. Each such transition's output is unique, and identifies which term has been found.

Transitions leading into the first state of each term have two-character labels; for `#SCHISM#' the first transition is labeled `#S'. This means that the first state is entered only after both characters in the label are matched in sequence. The first character is restricted to being a type code, i.e. `%',`#', `α', or `*'. The way to interpret the two-character label is that the first character is a precondition; it must be received to enable the transition into the first (or start) state of a term to be taken on receipt of the second character. This is a slight departure from the standard definition of the NFSA, in which each transition has only one label. This change reduces the number of states (two states with single character labels on incoming transitions are combined into one) and reduces the frequency of transitions out of the idle state.

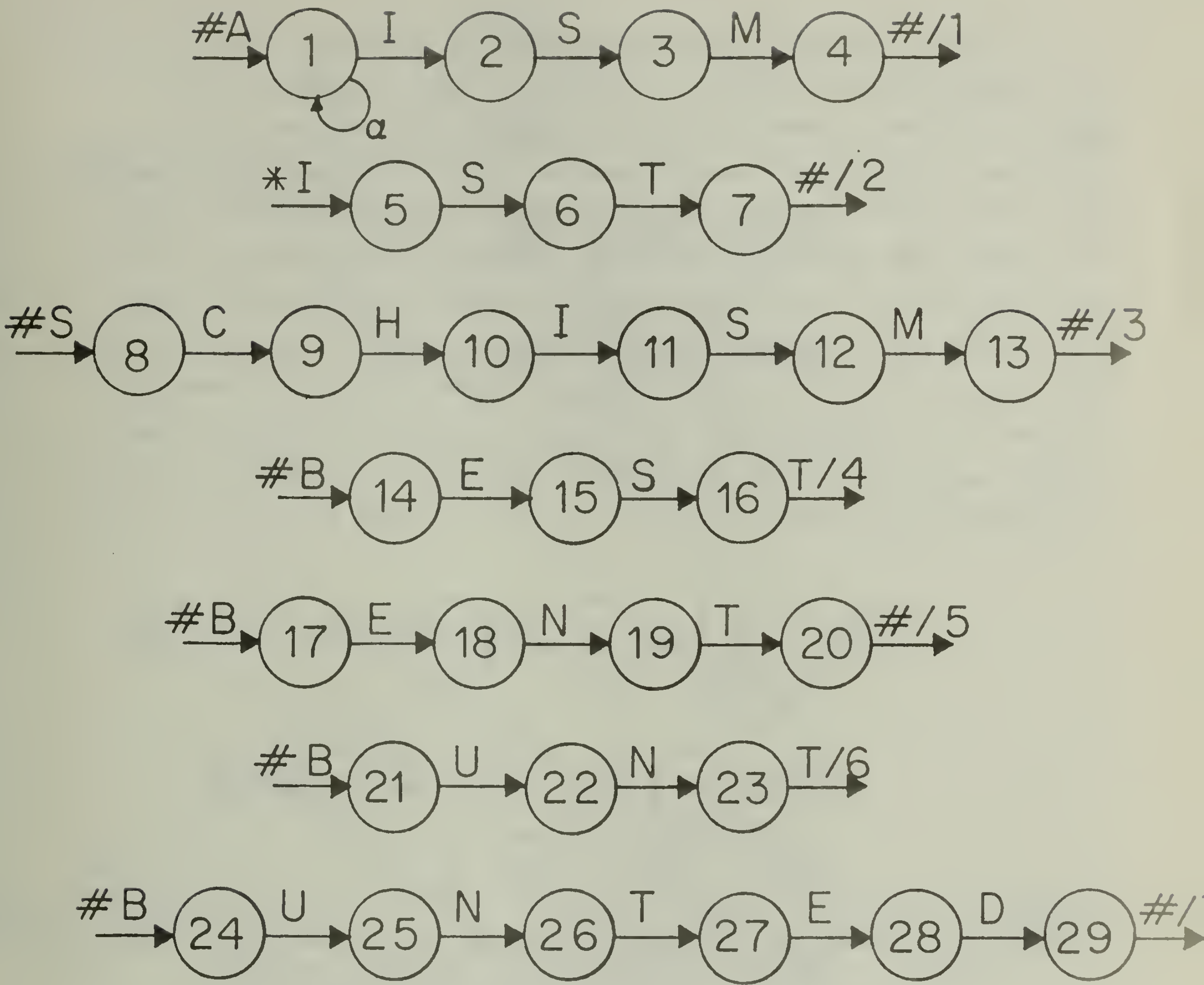


Figure 2.2 Diagram of Nonreduced NFSA State Table

Note the first state of \`#A?ISM#', that matches terms like \`atheism', \`antiterrorism', etc. The arc labeled \`α' means that the NFSA will stay in state 1 until it is forced out by a word separator. When an \`I' is matched, a transition is taken both to the next state <u>and</u> back to the present state. This is done because the first \`I' in the term may not be the start of \`ISM' (i.e. the first \`i' in \`antiterrorism'). This type of state is called a <u>loop state</u>.

Figure 2.3 shows the diagram condensed to eliminate multiple occurrences of term prefixes. For example, after recognizing `#BE', a fork transition is taken to states 15 and 17; one tine recognizes instances of `BEST' and another those of `BENT'. To simplify design of an NFSA term matcher, we will limit the maximum number of transitions out of fork states and loop states to three. Results to be presented later will show that this is a sufficient number to match practical sized tables. Even if the number of forks out of a state is greater than three, the table can be modified to limit fanout to three. Figure 2.4 shows the diagram for a table with four forks out of state 3. Figure 2.5 shows the diagram modified to limit the fanout. Obviously if backing the fork up to a previous state caused that state to have too large a fanout, the fork

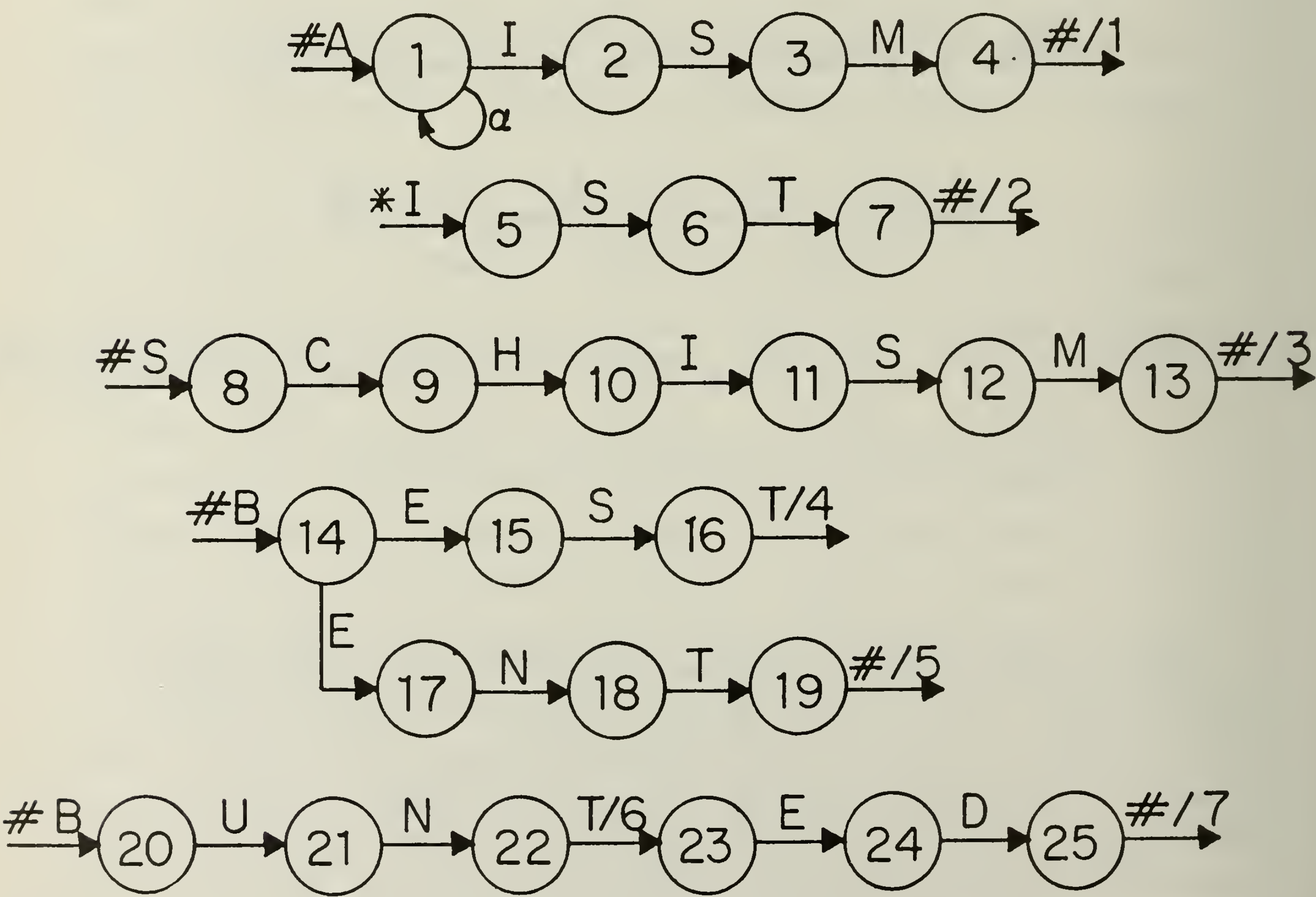


Figure 2.3 Diagram of Reduced NFSA State Table

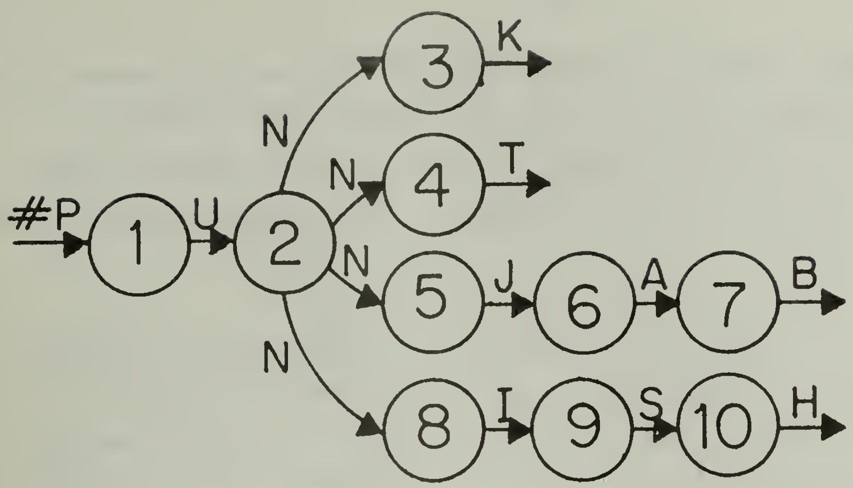


Figure 2.4 Diagram of Table with Fanout of Four

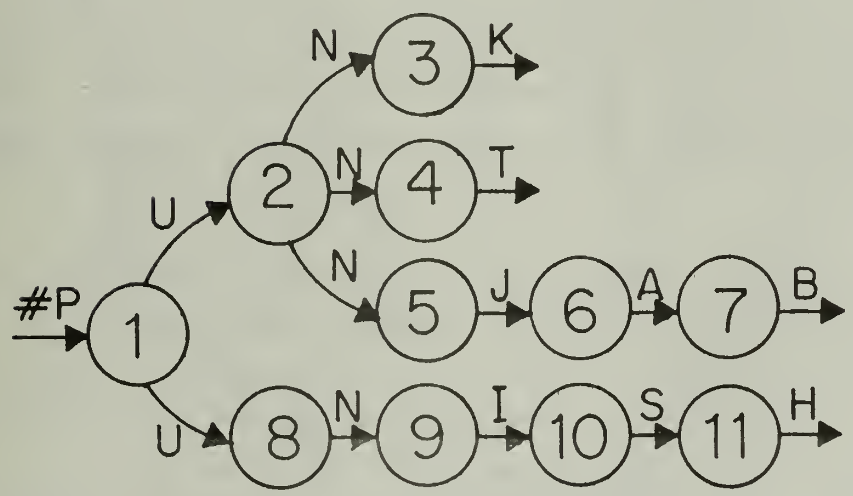


Figure 2.5 Table with Fanout Reduced to Three

could be backed up still further. If it were necessary to back a fork up past the beginning of a term, the excessive fork would just be entered in the table as a separate term.

2.1.1 Comparison of the FSA and NFSA Models

The NFSA model can be mapped into a hardware implementation having significant advantages over an FSA implementation, and this will be discussed in detail later. However, another advantage that the NFSA has

is that it is a simpler and more natural model of the search process. The NFSA state table is much simpler than the FSA table. It is easier to understand, contains fewer transitions, and is easier to generate from a list of search terms than its FSA counterpart. This is best shown by considering an example.

To illustrate the comparative complexity of the FSA model, consider the FSA state diagram corresponding to the list of terms in Figure 2.1. This FSA diagram is shown in Figure 2.6 As in the NFSA example, many of the transitions are not shown for clarity. In addition to the transitions shown, every state has a transition to state 2, which is taken if the input is a wordspace (`#′). Additionally, all states but 3, 4, 5, 6, and 19 take transitions to state 23 if the input is `I′. Finally, all states except 3, 4, 5, and 6 take transitions to state 1 (the idle state) if an input character is encountered other than one for which another transition is defined. These states do not have transitions to the idle state because they have transitions to other states defined for the entire input alphabet.

The diagram in Figure 2.6 has 25 states and 96 transitions. This compares with 26 states (counting the idle state not shown) and 65 transitions for the NFSA diagram in Figure 2.3. The fact that no backup transitions (i.e. from states 4 to 3 in Figure 2.6) are necessary in the NFSA table accounts for most of the difference in the number of transitions. Mismatch transitions to the idle state can be easily handled by the hardware, and it is not fair to count them as full-fledged transitions. Eliminating them from the accounting results in 75 match transitions for the FSA table and only 32 for the NFSA table. Each of these transitions has to be detected by analyzing the list of terms. Detecting some of the transitions (i.e. backups) requires extensive checking of states in one term against states in all others. This contributes to making the FSA table computationally more difficult to build than the corresponding NFSA table. Finally, consider how many comparisons must be done in the worst case match cycle. In state 2 of the Figure 2.6, the FSA must decide which of five non-default transitions to take (to states 2, 3, 7, 17, or 23). In Figure 2.3, the NFSA can be in at

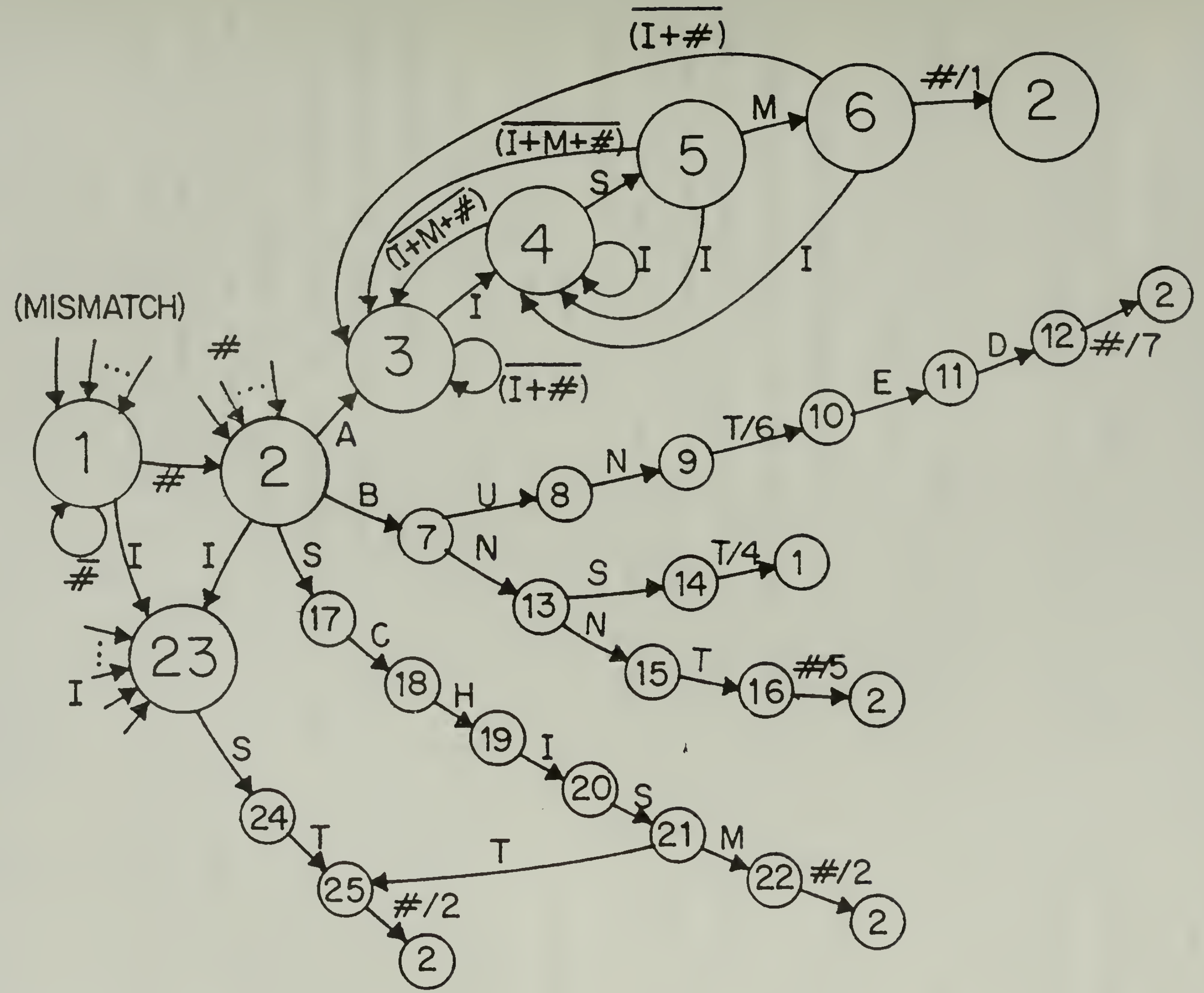


Figure 2.6 FSA State Diagram for Table to Match Terms of Figure 2.1

most three (non-idle) states simultaneously (1, 2, and 5). After receipt of the next input character, the NFSA must decide which subset of the three non-default successor transitions to take. It will be shown that the three yes/no decisions required by the NFSA requires simpler hardware to make than the five-way branch required by the FSA. Furthermore, it will be shown that this property holds for larger tables (i.e. the number of simultaneous yes/no decisions required by the NFSA grows relatively slowly with table size).

The advantage of using an NFSA for term detection follows from its ability to occupy several states at once; several potential search paths can simultaneously be followed with no need to back up in case of a mismatch along one of the paths. In a practical matcher, the number of states that can be simultaneously occupied must be limited to some small number, as each requires a separate state variable, yes/no comparator, and associated logic to effect state transitions. Simulations discussed later show that this number is small for practical-size state tables.

2.2 Implementing the NFSA Term Matcher with Multiple FSA's

Originally it was intended to use a number of small FSA's to interpret the nondeterministic state table. These would all share a common state table memory. Each FSA would follow a potentially successful search path. At each place where more than one path could be followed, an FSA would be activated from a pool of free ones, and would be started on the new path. This would be analogous to the NFSA `creating a copy of itself' in a successor state as stated in [HopU179]. Each active FSA would follow its assigned path either until a mismatch occurred or until the end of the term was found, whereupon it would be returned to the free pool.

Special logic is required to allocate FSA's from the free pool as new alternative search paths are encountered. Each new character can be the start of many new alternative search paths. These paths can be due to forks as was discussed above, or can be due to new terms whose beginning could be embedded in occurrences of other terms. Any or all of the free FSA's may get dispatched in one cycle, each FSA receiving a GO signal and

the state address at which it is to start. Either fast and highly parallel logic would have to be implemented to allow multiple allocations and startups in one character time, or synchronization logic (such as a FIFO buffer for input characters) would have to be used.

If a new search path appeared and no free FSA was available, some method of signalling the host to notify it that the table was too complex would be necessary. A problem that would have to be solved is signalling errors in a manner that would allow the host to recover from an error by eliminating an offending term from the table and then re-executing the search in multiple revolutions.

Contention among active FSA's for access to the state table memory could be quite severe. All active FSA's would need one memory cycle per match cycle. Methods exist for implementing multi-port parallel access memories with adequate bandwidth to support the needed number of FSA's, but these require too much logic for it to be practical to build such a memory for each term matcher in the system.

2.3 NFSA Implementation Using Memory-Mapped State Tables

Allocating FSA's to interpret a NFSA state table stored in a central memory resulted in several implementation problems. A far better way to approach the problem is to connect each FSA to its own dedicated memory and partition the state table so each state resides in the memory of one of the FSA's. Such a design is shown in Figure 2.7 It contains several small machines called character matchers (CM's), each corresponding to an individual FSA in the above discussion. A module called a Match Controller (MC) detects the occurrences of output transitions in the CM's, reporting instances of terms and the disk addresses at which they are found to the query resolver. A diagram of an individual character matcher (CM) is shown in Figure 2.8, and the format of the CM's startup and state table words appears in Figure 2.9

As was noted, it is not necessary to dynamically allocate processors (CM's) to follow currently active search paths. Instead, the state table is partitioned into groups of compatible states that can reside in a

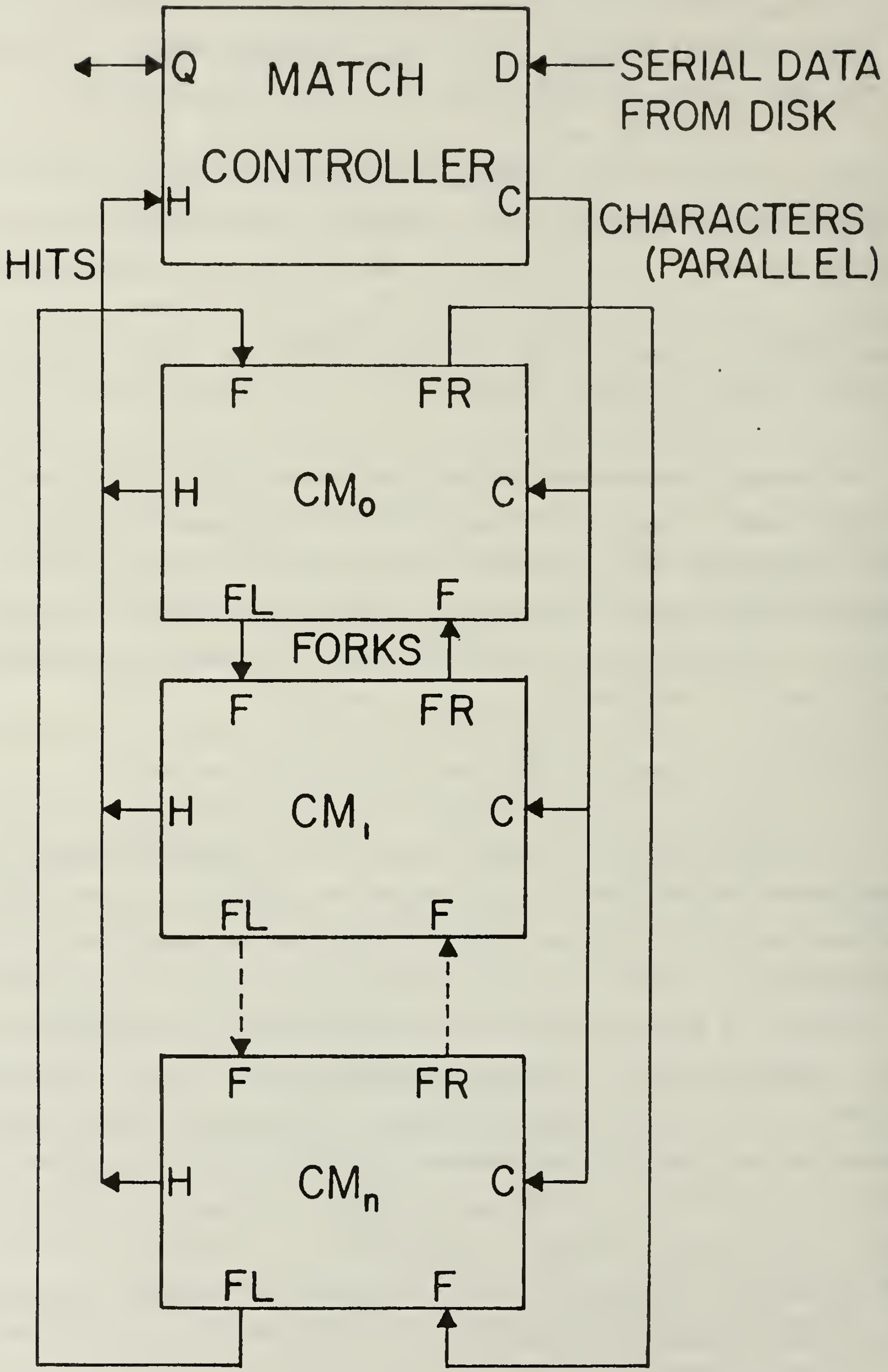


Figure 2.7 NFSA Term Matcher

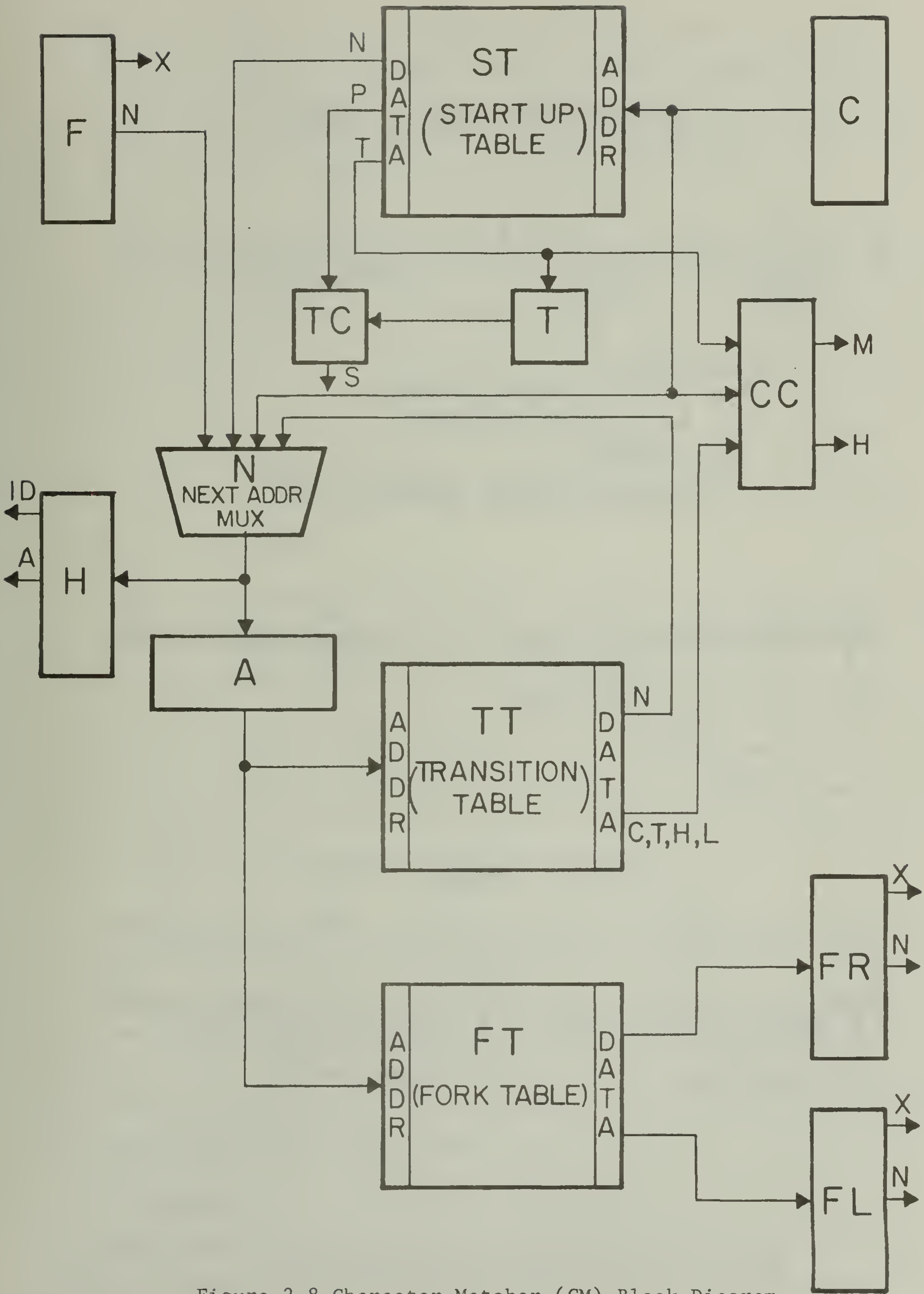


Figure 2.8 Character Matcher (CM) Block Diagram

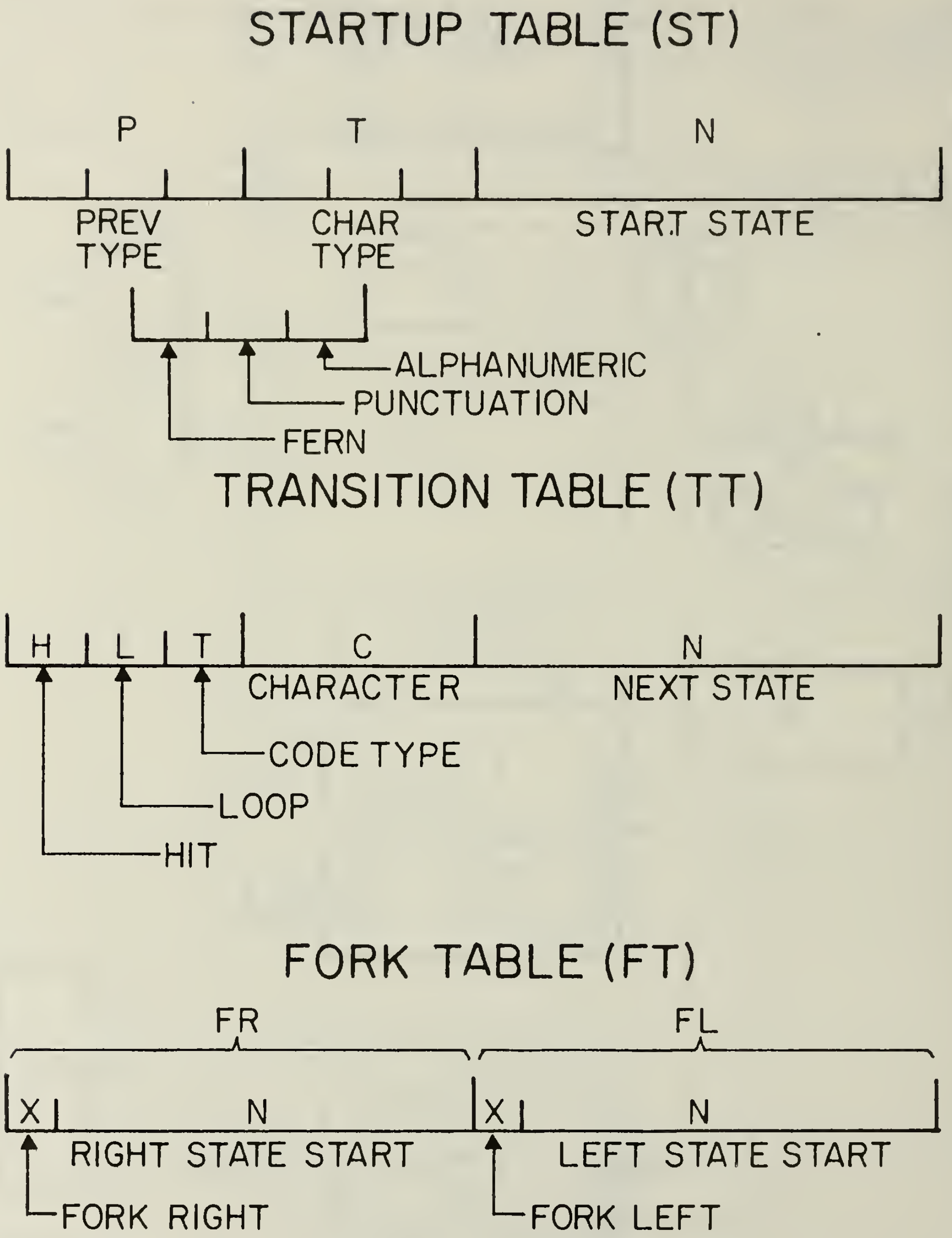


Figure 2.9 State Table Memory Word Formats

single CM's memory. Although compatibility will be more rigorously defined later, what it essentially means is that two compatible states can never potentially be occupied simultaneously. Partitioning states among CM's such that all states assigned to one CM are compatible assures that each CM will occupy a maximum of one state at any one time.

Figure 2.10 shows such a partitioning for the state table of Figure 2.3. The states I_0, I_1, and I_2 are idle states. The NFSA's idle state is split among the CM's, each one containing only transitions connnecting it to states assigned to the associated CM. A feature of the notation should also be pointed out: the two character startup label here has the first character of the label adjacent to the transition arc inside the idle state. This is more in line with the interpretation that the first character is a precondition for taking the transition after the second character is recognized.

This partitioning has the effect of `scheduling' the CM's at the time the table is built, eliminating the necessity of including logic to allocate CM's dynamically. Partitioning the state table, allocating states in advance to the CM's which will execute them, requires some extra computation. In effect, this computation is being traded for the opportunity to eliminate the hardware for scheduling the FSA's, and the memory contention resulting from many FSA's concurrently accessing one copy of the state table.

Introducing the idea of partitioned state tables allowed a very simple implementation for the CM's. Each CM is self-starting; no outside allocation hardware is necessary. During normal state transitions, each CM is only looking for one character code. This allows the comparator logic to be quite simple. No component of a CM gets cycled more than once per character match cycle. This allows faster cycle times to be possible with a given logic and memory speed.

One side benefit is that it can be determined when the table is partitioned whether it will fit in the available number of CM's. If not enough CM's are available, the search need not even be attempted; the table can be subdivided and the search performed in multiple revolutions.

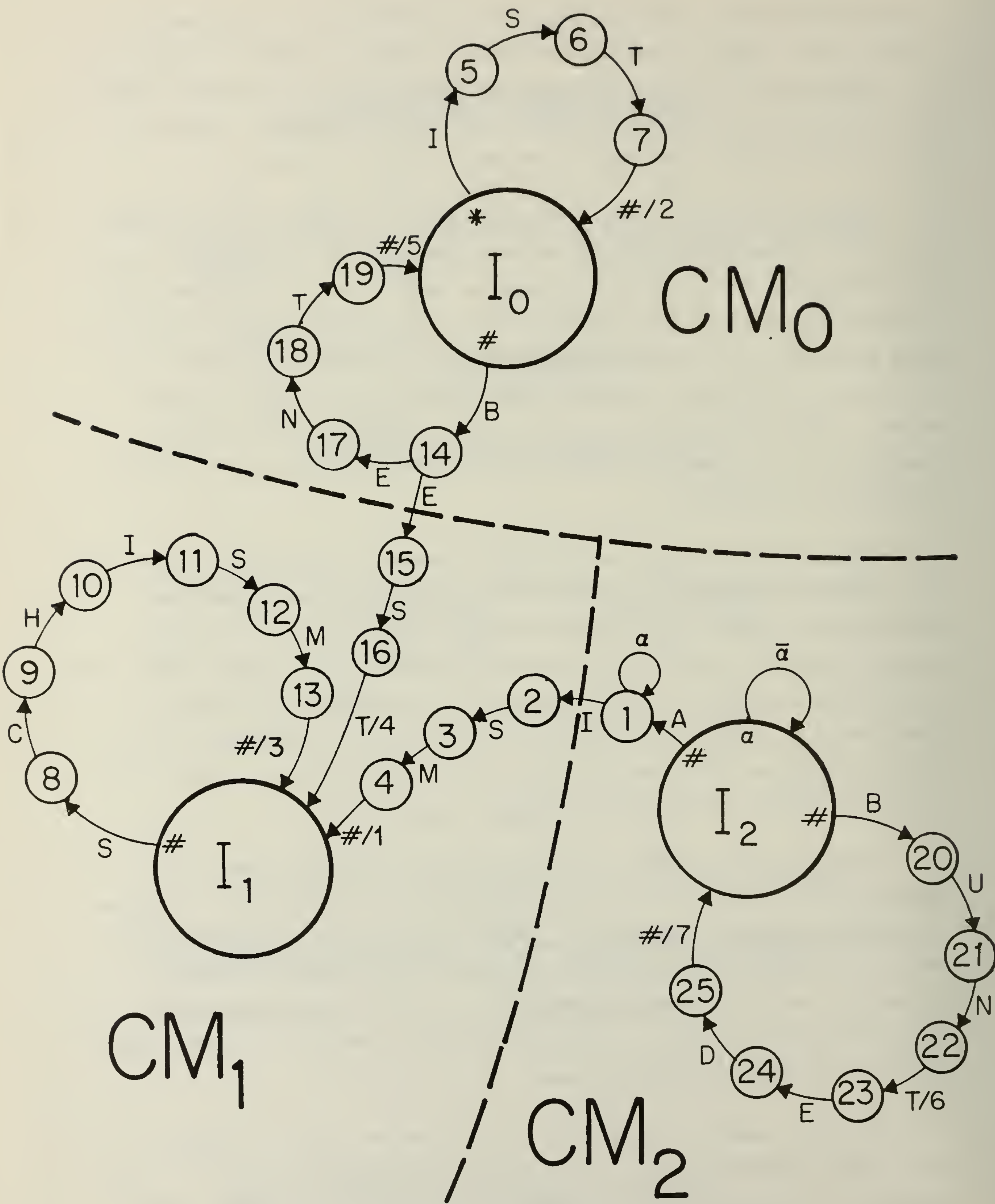


Figure 2.10 Diagram of Partitioned State Table

2.4 CM Operation

Each CM contains three RAM's: a startup table (ST), a transition table (TT), and a fork table (FT). The FT is mapped into a segment of the TT address space. The fork entries can be thought of as fields of a state's TT word, but since most states do not fork, the FT can be implemented more efficiently as a smaller, independent memory. The logic necessary to load the memories has been omitted from Figure 2.8 for clarity. The CM also contains the following registers:

- C - Holds the most recent character in the string to be searched.
- A - Holds the current state (TT and FT addresses).
- T - Contains the type ('%', '#', or 'α') of the character now in C.
- F - Buffer holding the startup state when the CM is being forked to by a neighbor.
- FR,FL - Holds fork table word contents, which are the start states for the right and left neighbor CM's when they are being forked to.
- H - Buffer for holding the terminal state and CM number when signaling that a term has been found.

Several of these 'registers' (F, FR, FL, H) are transparent - they do not need to actually latch any data, and can be implemented as buffers or even eliminated if the logic family being used to build the CM permits.

Additionally, the CM contains a character comparator (CC), a type comparator (TC, used to control startup), a next-state address multiplexor (N), and contains logic to control gating data among these components. The multiplexor N can also be eliminated if the logic family permits, instead being implemented as a bus with a selector enabling the source containing the next address onto the bus.

The state at address 0 in the TT is the idle state. Whenever a mismatch occurs and TT_1=0 for the current state, a transition is automatically taken to the idle state. The CM remains in the idle state until a <u>startup</u> transition occurs.

2.4.1 Startup

A CM can be started in one of two ways: either by being forked to by a neighboring CM, or by taking a startup transition defined by an entry in the ST. Inbound forks have priority over startup transitions, which in turn have priority over normal state to state transitions. To fork to a CM, one of its neighbors gates the start state into F (the state assignment prevents forking to an active CM). An example is the transition between states 14 and 15 in Figure 2.10. F_x=1 causes the multiplexor N to select F_n. The fork address is gated into A, and the corresponding state word is fetched. The CM is ready now to match the next input character.

A CM can also start itself in response to reception of a startup sequence. Startup sequences define transitions out of the idle state; it is therefore convenient to think of a CM as always occupying its idle state even though it may also simultaneously be in another state. Startup transitions work as follows: The contents of C for each input character are used as an address in the ST, and the corresponding word's contents are fetched. Each character's ST entry contains a previous character type (PCT) field (ST_p) which specifies what type(s) must precede the character to allow a startup to occur. Characters for which no startup transition is defined have a PCT of zero, which will not match any previous character's type. The PCT corresponds to the first character in the two-character startup labels mentioned earlier (which was restricted to being a type). If the TC detects a type match (ST_p & $T \neq 0$), it signals N to select the startup state field ST_n and gate it into A. The CM then starts as described for forks.

The previous character type restrictions allow terms to be started at word boundaries (ST_p=`#'), within words ($ST_p$=`α'), or anywhere (ST_p=`*'). Only one startup transition per CM is allowed for each character code. It must be remembered that startup transitions override normal state transitions. It is necessary that the state assignment insure that startups do not interfere with normal matching. Sometimes, however, it is necessary to use a startup transition to intentionally drive a CM out of a

particular state. For example, a startup transition is used to force exit from an EVLDC loop state at the end of a term.

2.4.2 CM State Transitions

When a state is entered by gating its address into A, the contents of the corresponding word in the TT are fetched. At the start of the next cycle, the character code field in the TT word (TT_c) is compared with either the input character in C or the current character type in ST_t, based upon the value of TT_t.

If the character or type matches, first (if the hit bit TT_h=1) the hit register H (which contains the present state address) is output to the Match Controller. Then, the next state (in TT_n) is gated into A and H. If there was a mismatch, and if the loop bit TT_l=0, A is cleared to force a transition to the idle state.

The loop bit being set forces TT_n to be used as the next state address regardless of whether or not the character stored in the state table entry matches the input. TT_l is called the loop bit because TT_n of states with this bit set will usually contain the address of the state itself, causing the CM to loop in the same state until it is forced out by a startup transition or by being forked to (i.e. state 1 in Figure 2.10). However, in some cases a loop state's TT_n field may contain the address of a different state. Such a case will be discussed in Section 2.4.6.

2.4.3 Forking

A CM can start either or both of its neighbors by forking to them. Most tables have many fewer fork states than normal states, so fork states are allocated only the top 1/8 of the state-table address space. Whether or not a state is a fork state is determined by its address.

Fork states have normal transition sequences which work exactly as described above. Forking is done in parallel with normal transitions. If the CM is in a fork state, if the input character matches successfully, and if one or both of the fork enable bits (FR_x, FL_x) are set, then the corresponding fork addresses in FR and FL are gated out to the neighboring

CM('s). The neighbor then starts as described in Section 2.4.1. Forking is done in the following situations:

1. Incompatibilities - If a term is being matched in a CM and one of its states is incompatible with another in the CM's memory, control must be transferred to another CM to match the incompatible state. To do this, the last compatible character is made a fork state, and (if reached) a fork is made to a neighboring CM. For such states, TT_n contains the address of the idle state.
2. Terms with common prefixes - The fork is made after matching the last common character (i.e. state 14 in Figure 2.10). The CM proceeds to look for one alternative, and the neighbor(s) look for the other(s).
3. EVLDC's - After matching the prefix, a loop state is entered which will fork to another CM each time the first character in the suffix is encountered (i.e. state 1 in Figure 2.10). Loop transitions are described in more detail below.

2.4.4 Loop Transitions

Any state with its loop bit TT_1=1 is called a <u>loop state</u>. An example is state 1 in Figure 2.10. Loop states are used to eliminate the need to back up in case of a mismatch, as was necessary in the FSA. If TT_1=1, the mismatch transition back to the idle state is inhibited. The next state address in TT_n is gated into A regardless of the result of the match. TT_n usually contains the loop state's own address (hence the name `loop state'), although Section 2.4.6 will discuss an exception. In addition to remaining in the same state, loop states fork to a neighboring CM to continue matching the suffix. They therefore must reside in the fork state portion of the address space. The CM remains in the loop state even after forking. This insures that if the suffix match fails, the CM will still be looking for a correct instance of the suffix. Thus, at least two CM's are required to match an EVLDC term, one to match the suffix and one to continue monitoring the input for each potential start of the suffix string.

In the absence of any further outside influences, the loop state would remain occupied forever. The CM can only be forced out of the loop by being restarted or by being forked to. Since it is usually desired to stop looking for the suffix at the end of a term, it is necessary to include a startup transition in the ST so that term separators force the CM back to the idle state. This special startup transition is called a kill transition. One appears in Figure 2.10, going from I_2 back to I_2. Its label, `$\alpha\bar{\alpha}$', corresponds to any alphanumeric followed by a non-alphanumeric, and is equivalent to having both the transitions `α%' and `α#' in the CM. It would be implemented by including an entry in the startup table for every non-alphanumeric, each entry having $ST_p$=`α' and each going to the idle state. Each time a non-alphanumeric was seen following an alpahnumeric, the loop would be killed.

Actually, loop transitions are very powerful and can be used for several purposes in addition to matching EVLDC's. Figure 2.11 illustrates the use of a loop state to match a term containing two words separated by a variable number of word separators. At the end of the first word (`PUNK'), the loop state is entered. The CM remains in that state until the start of `ROCK' is seen, whereupon a fork is done to match `ROCK'. A kill transition must be included for every character not having a startup transition, except word separators. The one for Figure 2.11 is implemented by setting the ST entry for all characters but `P' and word breaks to force the CM to the idle state.

2.4.5 Recognizing Document Formatting Codes

The query resolver needs to be notified when document formatting codes (ferns) are encountered. Actually, these are recognized just like one-character terms. One CM is assigned to matching ferns, and has a startup sequence defined for each. Figure 2.12 shows an example. Since a startup transition cannot cause a hit to be reported, it must go to a second state. This state reports a hit and takes a transition back to the idle state regardless of the next input. This can be accomplished by setting TT_h=1, TT_t=1, TT_c=`*' (all type bits set), and TT_n=0 (the idle state address). Even if the character after the fern causes a startup

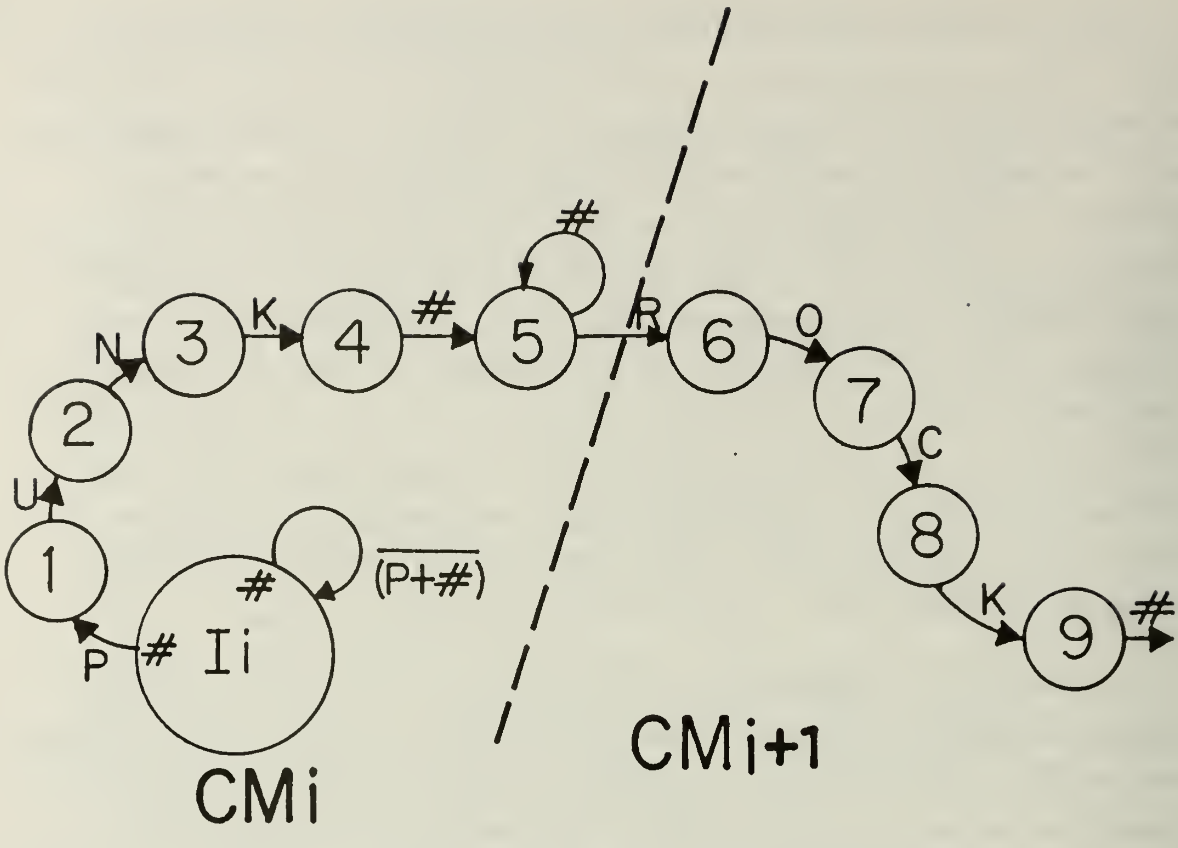


Figure 2.11 Loop States Used to Match Word Breaks

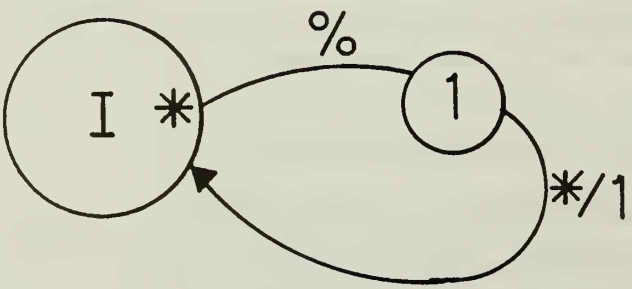


Figure 2.12 Transitions to Match Context Boundary Codes

transition to be taken, the operation of the CM will insure that the hit is reported. Depending upon the requirements of the query resolver, each unique fern code would probably force the CM to a different state so the fern code that was received could be deduced from the output.

2.4.6 Bounded EVLDC's

State 1 in Figure 2.10 shows the usual type of embedded variable length don't cares, where the length of the don't care string can be of arbitrary length. The only way the CM can be stopped from looking for the suffix is to force it out of the loop state with a kill transition (i.e. when a word separator occurs). Sometimes, however, it might be desired to match strings with a variable, but bounded, number of don't cares. A table to match such a string is shown in Figure 2.13 This table will match \`AB', \`A*B', or \`A**B'. The loop bit $TT_1$=1 for states 1 and 2. The match character for states 1, 2, and 3 is \`B'. When the initial \`A' is seen, $CM_0$ enters state 1. For the next two input characters, instead of mismatches sending $CM_0$ back to the idle state, the CM will go to the next state regardless of the input. If a \`B' is seen while CM_0 is in states 1, 2, or 3, a fork to state 4 (in CM_1) will be performed. State 3 returns to the idle state regardless of the input.

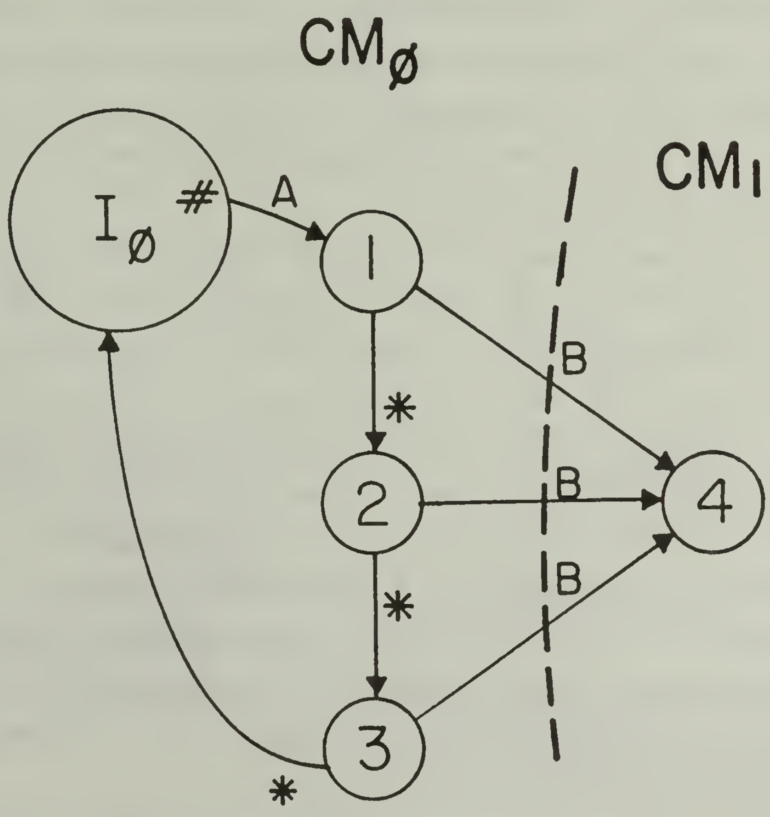


Figure 2.13 Bounded EVLDC

2.5 MC Operation

Figure 2.14 shows a block diagram of the Match Controller (MC). The MC is arranged around a 16-bit internal bus, to which the following components are connected:

1. Four registers (F, A, D, and S) for communicating with the query resolver (QR). The F register allows the query resolver to specify the matcher's function, and S displays the matcher's status. Registers A and D are set by the query resolver to control the address and data (respectively) when loading data into the CM's.
2. Two buffers (C, D) and one register (H) for communicating with the CM's. Commands and match characters are transmitted to the CM's over C, data to be loaded into the CM memories is transmitted over D, and hits (successful matches) are reported by the CM's into H.
3. A small FIFO (Q) for reporting hits to the query resolver. The FIFO is used because sequential characters can cause multiple hits which must be processed by the query resolver. Buffering these in a FIFO allows more flexibility in building the query resolver hit processing logic.
4. A serial/parallel code converter to handle data from the disk. The serial data is converted to parallel format, then the bytes are looked up in a ROM to convert them to the character set used by the CM's. The ROM allows the character set size and codes to differ between the CM's and the storage medium. Case conversion can also be done if desired. A RAM could be used to allow these functions to be programmed, but the extra complexity involved in loading a RAM was thought to outweigh any benefits.
5. A counter to generate addresses corresponding to characters coming from the disk. The counter allows each hit to be accompanied by the address (displacement into the region) at which it occurred.
6. A timing generator to allow manufacturing timing signals for the MC and CM's from the disk bit clock and character strobe signals.

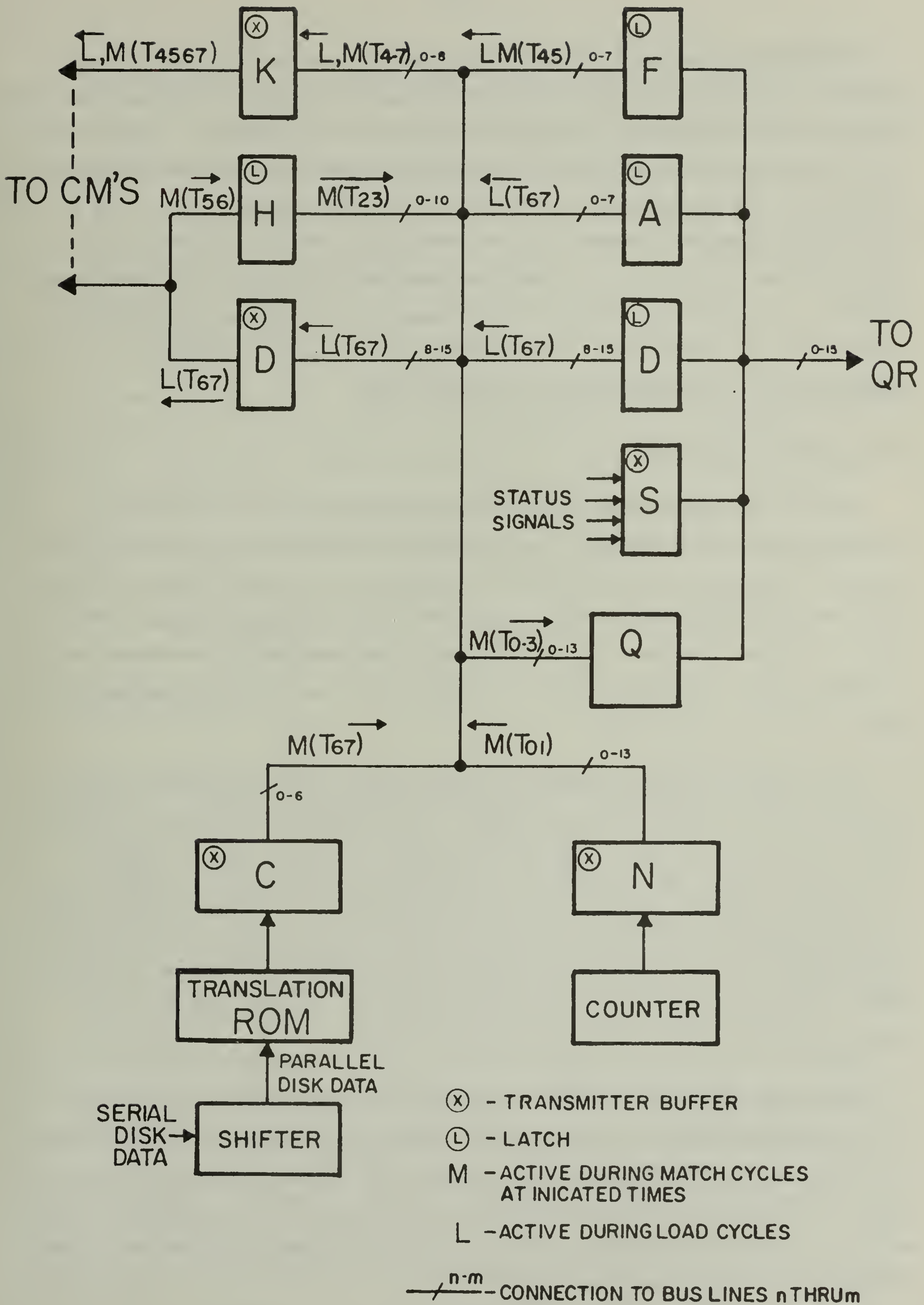


Figure 2.14 Match Controller (MC) Block Diagram

The MC also contains a small state machine to sequence gating data onto the internal bus. The MC is interfaced to the query resolver over an external bus, to which F, A, D, S, and Q are also connected. The decision to use the query resolver to load and control operation of the matcher was made because it is anticipated that the QR will be implemented using a small general purpose computer (i.e. a microprocessor), and it is expected that programming the QR to support controlling the matcher will not be a problem. If the QR is eventually implemented using different technology, the matcher and the QR might be interfaced to a small control microprocessor to oversee both their functions.

2.5.1 Timing Generation

Each cycle (character time) is broken into eight equal segments, corresponding to the eight bit-times during which data is received from the disk. The basic clock signal is derived from the bit clock from the disk, which is nominally around 100ns for 3330 technology drives. The eight bit times are named T0 thru T7, T0 corresponding to the first bit of each data byte. All clock times in both the MC and CM′s are defined in terms of this nomenclature.

The timing generator provides three signals of single-time duration (T1, T3, and T7) used solely as clocks, six signals of two-time duration (T01, T23, T45, T56, T67, and T70) used both as clocks and to control gating data, and one signal of four-time duration (T0123) used as a gate. To clarify the nomenclature, T01 is one pulse occurring during T0 and T1 of a cycle, T70 is a pulse occurring during T7 of one cycle and T0 of the next, and T0123 is one pulse occuring during T0, T1, T2, and T3 of a cycle.

All data transfers in both the MC and the CM′s are set up so that data is enabled onto a bus during (at least) two successive times, and is clocked into the receiving register by a clock signal occurring in the middle of the interval during which it is available. For example, if data were gated onto a bus during T67, it would be clocked into the receiving register on the leading edge of T7.

Three timing signals are provided externally for use by the CM's (Ta, Tb, and Tc). Ta corresponds to T56, Tb to T70, and Tc to T0123, although special logic inhibits them during certain cycles. Figure 2.15 shows a timing diagram of these signals, and also illustrates when data is available on the I/O registers of the CM's, and when it is clocked into them. Tc is used only during load cycles; its function is to strobe the write enable line of the memory being loaded.

2.5.2 Load Cycles

To load the CM's, the query resolver places the CM memory address and data into the F, A, and D registers. The MATCH bit in F is cleared to indicate a load cycle. During T45, F will be gated onto the MC bus and out to the CM's via C. Each CM will use Ta to strobe C into itself. During T67, A and D (the memory address in the CM and the data) will be gated onto the MC bus and out to the CM's via C and D. If it was the one addressed, the CM will use the address and data strobed into C and D by Tb to load the addressed byte of memory during Tc.

2.5.3 Match Cycles

For the moment, details involved with starting and ending match operations on a data stream will be ignored. It is assumed that the match cycle being described is somewhere in the middle of the data stream (i.e. in the middle of a track). If the data flow through the complete term matcher (MC and CM's) is considered, the system can be regarded as a pipeline. For a given character, bits arriving serially from the disk are assembled into a byte during one cycle, and code conversion is done by the ROM during the next cycle. During T45 of the second cycle, F (the function register) is gated onto the MC's internal bus, thru C, and from there broadcast to all CM's. The MATCH bit informs them that this will be a match cycle as opposed to a load. During T67 of the second cycle, the ROM contents (the translated character) are similarly gated onto the bus and through C to all CM's. During the third cycle, the active CM's compare the character with TT_c for their current states, and during T56 of that cycle, any hit is reported to the MC, during which time it is strobed into the MC's H-register. If there was a hit, then during the fourth

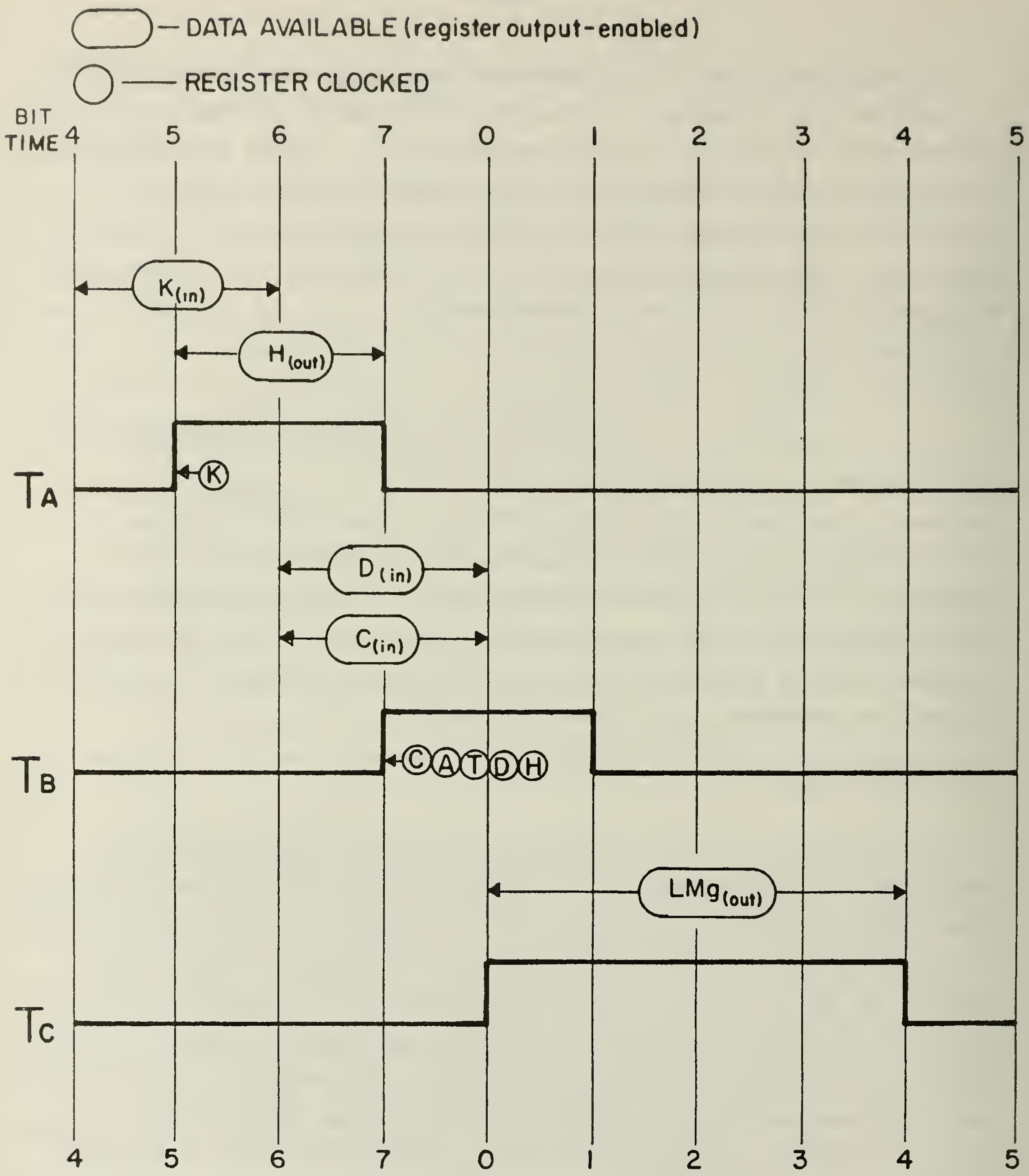


Figure 2.15 Character Matcher (CM) Timing Diagram

cycle the character's address is gated from N (the address counter) onto the MC's internal bus during T01 and is strobed into Q (the FIFO), and during T23 the terminal state address (in H) is gated onto the bus and into Q. Sometime later, the hit will be read from the FIFO by the query resolver.

When N (the address counter) runs out at the end of the track, the MC gates it and H (which contains nothing in particular) into Q. The query resolver, seeing the 0 in the address field of a hit, can detect the end of the track. The counter going to 0 also inhibits any further match cycles. Notice that due to the delay in propagating a character through the matcher, the address in N must be displaced by a fixed value so the count runs out at the proper time. If an up-counter is used to match an n-byte data stream, N must be loaded with -(n+3) at the start of the track.

The disk must continue to provide clock pulses even between tracks so timing signals can be generated for load cycles. Most disks have clocks which free run at approximately the bit rate even when data is not being transferred, so this is not a problem.

2.6 Matching a Data Stream

Figure 2.16 shows a timing diagram representing all major signals in the term matcher occuring during matching of a three-byte data stream. A track length of three bytes is sufficient to show the special cases involved in start of track and end of track processing, and also is sufficient to illustrate the propagation of data through the system.

For purposes of the example, assume that the CM's have been loaded with a state table, that a 0 was strobed into C in all CM's to force them to the idle state, and that MATCH in F was set. The matcher is waiting for the INDEX pulse to arrive from the disk, signalling start-of-track. Further assume that the first three bytes of the track will cause one of the CM's to generate a hit. It may be helpful to refer back to Figures 2.5 and 2.11 for review of the names of the components of the MC and CM's.

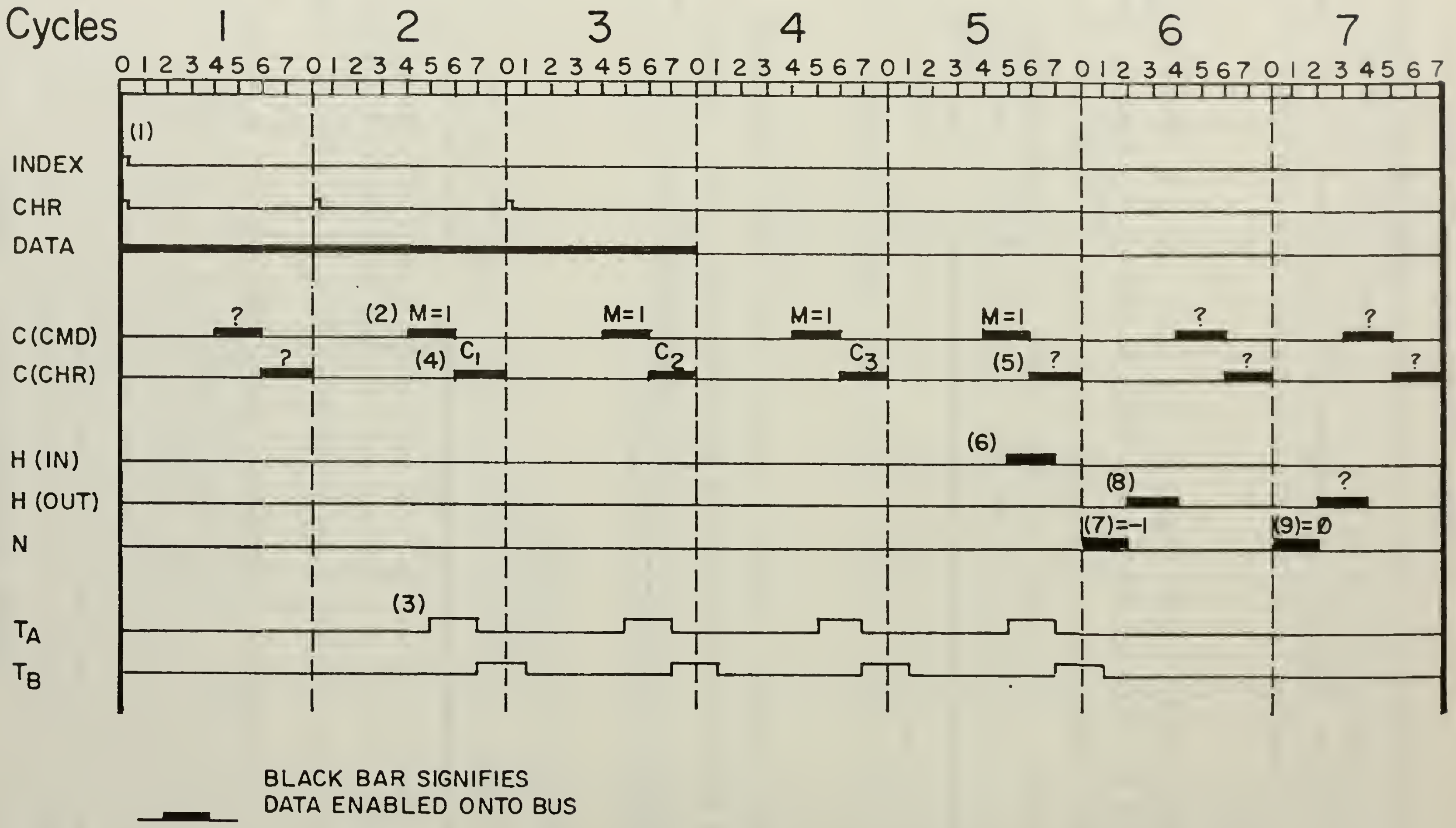


Figure 2.16 Timing Diagram for Three-Character Match Sequence

During T0 of the first cycle (character time), INDEX and CHR co-occur ([1] on Figure 2.16), signifying the arrival of the first bit of the first character on the track. During the next eight bit times, data bits arrive from the disk and are assembled in a shift register in the MC. The contents of F and the ROM (whatever they happen to be) are gated through C to the CM′s during T45 and T67 of the first cycle, but since Ta and Tb are inhibited during the first match cycle, the CM′s ignore them.

At the start of T0 of the second cycle, the new data byte is gated from the shift register to the ROM address register. During T45 of this cycle, F (the function, containing the MATCH bit) is gated through C to the CM′s [2], and since Ta occurs during this cycle [3], it is gated into each CM. MATCH going from 0 to 1 clears T in each CM, and enables the CC′s. During T67 of this cycle, the first data byte ([3] in Figure 2.16) is sent from the ROM to the CM′s via C. Tb occurs during T70, causing the character to be strobed into C. The CM′s will remain in the idle state, and the character will be looked up in the ST of each CM.

During the third cycle, the serial data for the third character is received from the disk and the translated version of the second is sent from the ROM to the CM′s (during T67). At T7, the occurrence of Tb strobes the third character into C, and strobes the type of the second character (from the ST_t field of the ST entry, looked up during the preceding cycle) into T in the CM. Tb also gates a new state address into A, but since there is no match and no startup sequence (T was 0, so the TC did not generate a startup signal), a 0 is again gated into A. Thus, the CM′s remain in the idle state for another cycle.

During the fourth cycle, no data is received from the disk. The third byte is sent from the ROM to the CM′s and strobed into C by Tb. During this cycle, the ST entry for the second character is fetched, and the TC compares its ST_p field against the type of the first character, which is now in T. By assumption, they match, and a startup signal is generated. The next state address is taken from ST_n, and is clocked by Tb into the address register A.

During the fifth cycle, again no disk data is available, and furthermore the data sent from the ROM to C is not defined ([4]). By T7 of the fifth cycle, when Tb occurs, the TT word for the state entered after receipt of the second character will have been fetched. The character code field (TT_c) will have been compared against the third data byte, and (by assumption) a match will have been generated. Since the hit bit TT_h=1, a hit will be generated, and Tb will gate the state address and CM number out of the H-register in the CM into that of the MC.

During the sixth cycle again no valid character is sent to the CM's, and in fact what they do is of no further interest. Since a hit was received by the MC during the previous cycle, the hit address (-1) is gated from N onto the MC bus and into Q during T01 ([5] in Figure 2.16). The hit address in H is gated into Q during T23 ([6]). The counter N will be counted at T6, and will overflow. This will prevent any further Tb signals from being generated, so no further (spurious) hits will be reported by the CM's.

During the seventh cycle, since N contains a 0, its contents are gated onto the bus and into Q during T01. The meaningless contents of H are gated into Q during T23. The overflow of the disk address counter N at this point sends the MC out of match mode, and no further action takes place without intervention by the query resolver.

Eventually, the data for the two matches (one for the character sequence, the other for end-of-data) will appear at the output of the FIFO, and will be processed by the query resolver. During the succeeding interval, the disk is positioned to a new track, a new table is loaded, the CM's are forced to the idle state, and the match is performed on a new cylinder.

2.7 MSI Prototype of the NFSA Term Matcher

As anyone with experience in logic design will attest, the initial block diagram of a system will usually bear little resemblance to the final implementation. To be truthful, the ones shown in previous sections (Figures 2.5 and 2.11) were modified many times during the NFSA matcher's

gestation period. If the design is only taken to the block diagram stage, problems often tend to be ignored or swept under the rug. To iron out the details of the design and to demonstrate its feasibility a prototype of the NFSA Term Matcher was designed using MSI circuits. It is configured with an MC and four CM's, enabling it to handle most single-query searches. It will be available for use in an experimental retrieval system to aid in further research.

2.7.1 MSI Match Controller

Figure 2.17 is a reproduction of the logic diagram of the MSI Match Controller. It consists of 41 IC packages, most of them being MSI registers. All but one of the packages are standard 7400-series TTL, the exception being the ROM, an Intel 2716 2Kx8 bit EPROM. The MC is connected to the QR via 21 lines, to the CM's via 21 lines, to the disk via 5 lines, and to V_{cc} and ground through another two (a total of 49).

The INDEX line from the disk signals the start of data on a track. Assuming the MC has received a MATCH command from the QR (on the X_7 line into F), receipt of INDEX places the MC into match mode. CHR synchronizes the timing generator to characters arriving over the DATA line; individual bits are clocked by CLK, which also clocks the Johnson counter in the timing generator. A truth table for the Johnson counter appears at the left side of Figure 2.17, next to the counter. Bits arriving on the DATA line are shifted into the 74164, and at the end of the cycle are strobed into the 74175 which serves as an address latch for the 2716 ROM. Output data from the ROM (the translated, six bit versions of the input bytes) are gated onto the internal bus at the appropriate time through the 74244 tri-state driver.

The disk address counter (N) appears just above the deserializer/code converter, and is implemented using four 74161 counters. The system will be used on fixed-length tracks, so the initial count is set using jumper wires. The 74244's gate the count onto the bus at the appropriate times after a hit is detected and after the counter overflows. The carry out of the high order counter is OR-ed with the ERROR line from the disk, and the result is used to take the MC out of match mode.

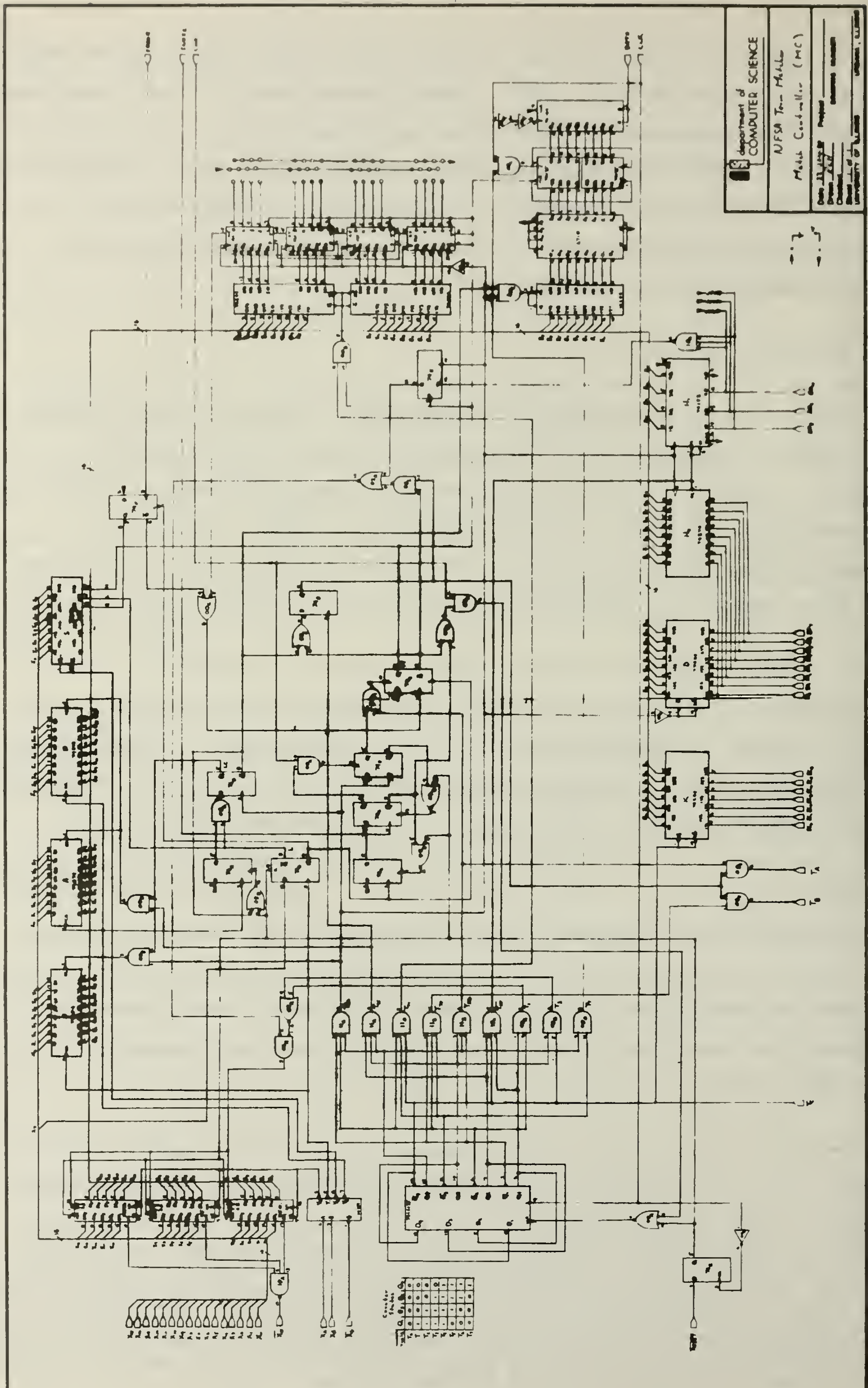


Figure 2.17 Logic Diagram of MSI Match Controller

Interface to the QR is through three 74374 tri-state latches (F, A, and D), one 74244 buffer (S), and three 74225 FIFO registers connected in parallel. The FIFO′s accept hit record data off the internal bus (B_0-B_{15}) and output it onto the external bus (X_0-X_{15}). F, A, and D accept data from the X-bus and output it onto the internal B-bus. The 74139 decoder controls gating data between these devices and the X-bus in response to the X_a, X_b, and X_g signals from the QR. RESET, also from the QR, must be asserted (low) between power-up and receipt of the first load or match command.

Buffers C and D (74244′s) and register H (one 74374 and one 74173) control gating data (commands, match characters, load data and addresses, and hit reports) between the internal bus B and the CM′s. Commands, characters, and load addresses are output through C, load data is output through D, and hit reports are input into H. Since their transactions do not overlap, H and D share the same lines connecting them to the CM′s. Finally, clock signals to the CM′s are output over Ta, Tb, and Tc.

Gating among these components is controlled by a state machine implemented with 7474 D-type flip-flops. These keep track of whether the MC is in load or match mode, sequence the loading of data into the CM′s, and control progress through the match (waiting first for INDEX, then waiting the required number of cycles before enabling the timing signals to the CM′s, and finally shutting the machine down when N overflows or an error is detected).

2.7.2 MSI Character Matcher

Figure 2.18 shows the logic diagram for the MSI Character Matcher (CM). Most of the 35 packages are 256x4 bit RAM′s and MSI buffers and latches. The remainder of the packages are other MSI functions (comparators, encoders, decoders, and multiplexors). Very little random logic is used, in fact there is only one 7400 NAND-gate package and one 7404 inverter package. No flip-flops are used; actually there is no sequential control logic whatsoever. As with the MC, most of the logic for the CM is 7400-series TTL, the only exceptions being the RCA MWS5101AEL3 256x4 bit static RAM chips.

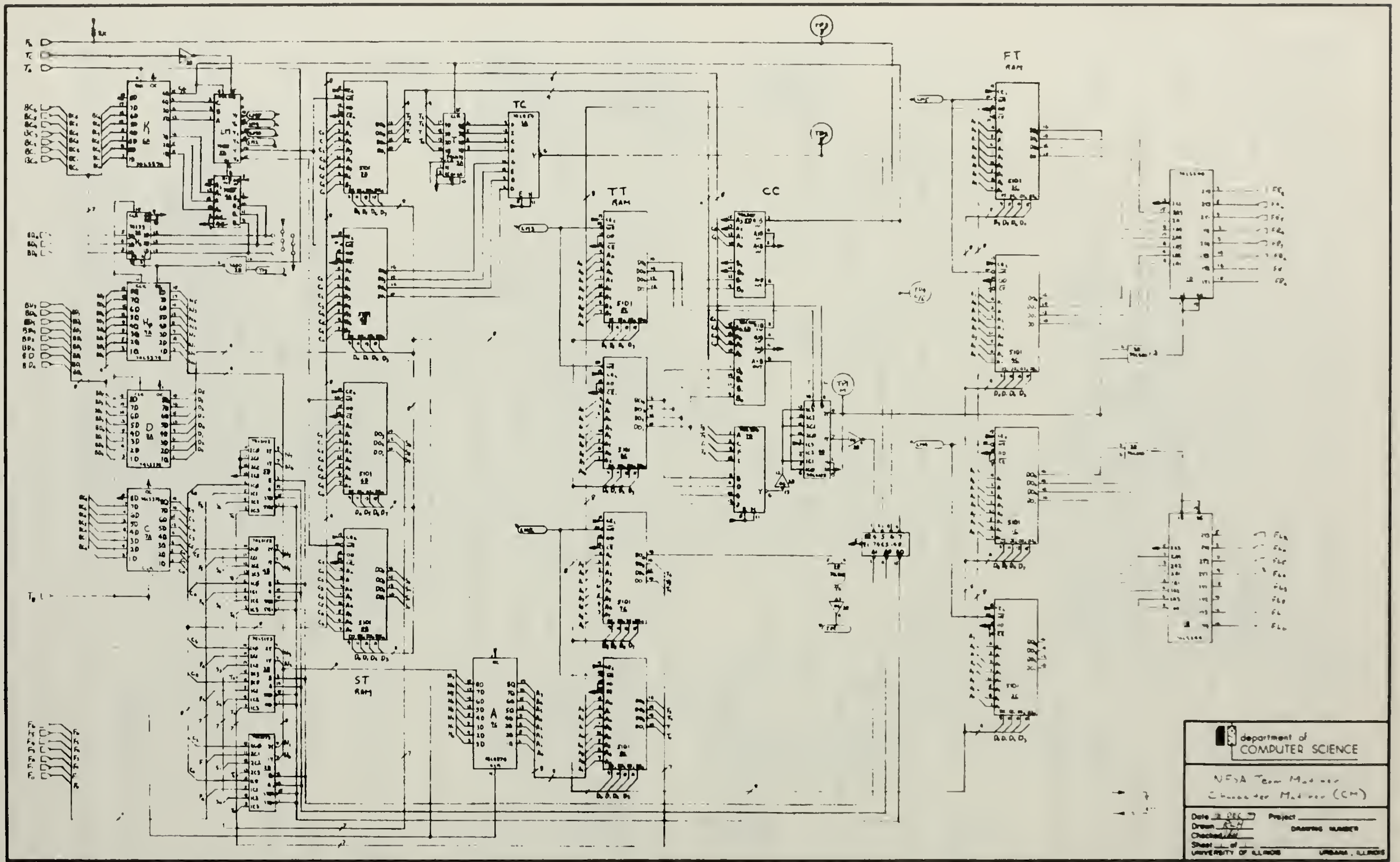


Figure 2.18 Logic Diagram of MSI Character Matcher

As mentioned, 21 lines are necessary to connect the CM's to the MC. Additionally, each CM has 24 lines connecting it with its neighboring CM's to handle forking. With one line for V_{cc} and another for ground, the total number of I/O pins for the CM is 47.

The ST, TT, and FT memories require four 256x4 bit RAM chips each, a total of 12 packages. The ST address is taken directly from the C register (character input) and the TT and FT address is taken from the A register, which is connected to the next-state multiplexor via lines N_{0-7}. The TT and FT are therefore really the same memory in this implementation; since the 5101's are only available in 256-nibble packages there was no way to use less memory for the fork tables. In this implementation, therefore, every state can be a fork state.

The TT and FT address can be taken from one of four sources, selected by multiplexor N (implemented using four 74153's). Which source is selected is controlled by the 74148 priority encoder appearing near the center of Figure 2.18, between the TT and FT memories. The priority scheme gives highest precidence to load cycles, followed by forks, startups, and matches in that order. During a load cycle, the address is taken from C (allowing the ST, TT, and FT memories to be loaded in the same manner by the MC). If a fork is being done by a neighboring CM (F_x=0) the TT and FT address is taken from the F_{0-6} inputs. During startup sequences, A is loaded from the ST (S_{0-6}), and during match transitions it is loaded from the TT (T_{0-6}). If none of the above conditions holds (i.e. for mismatches returning the CM to the idle state), none of the multiplexor inputs are selected and A is loaded with a zero. Zero, of course, is the address of the idle state.

Loading a CM is accomplished by placing the CM number and memory select code into the K register, along with MATCH=0 to signal a load cycle. The address within the memory to be loaded is placed into C, and the data is placed into D. Since the RAM chips are four bits wide, two are loaded in parallel during a load cycle. For example, memory select code 0 loads the low order 8 bits of an ST word (the startup address field); code 1 would cause the high 8 bits of the word to be loaded. The

D-register is broadcast to the data inputs of all memories, and the C-register is wired directly to the address inputs of the ST memories and is fed through N and A to the TT and FT memories during a load cycle. During Tc, if the CM number agrees with that jumpered into the inputs of the 7485 comparator (adjacent to K on Figure 2.18), the 74138 decoder will strobe the write enable line of the memory chips to be loaded, causing the data to be written into them (the chips are always selected).

If a match occurrs in a state having one of the fork enable bits FR_x or FL_x set in the FT word, a fork to a neighboring CM is done. The FR lines of a CM′s left neighbor and the FL lines of a CM′s right neighbor are connected to the CM′s F input lines. When one of the neighbors executes a fork, its fork output buffer is enabled, pulling the F_x input of the forked-to CM to 0. This causes the 74148 priority encoder and the 74153 multiplexors (N) to clock the F input lines into address register A during the next Tb. It is up to the state assignment routines (see Chapter 3) to insure that both neighbors do not simultaneously attempt to fork to a CM.

A startup transition occurs whenever one of the bits set in the ST_p field agrees with the corresponding bit in the T register. When the TC (a 7454 and-or-invert) detects that the required previous type agrees with the type of the previous character, the 74148 encoder forces the multiplexor to gate the startup address in ST_a into A.

The character comparator (CC) compares the character field of the TT (TT_c) with either the current character′s type in ST_t (if TT_t=1) or with the current character itself (if TT_t=0). For type compares, a 7454 is used as in the TC. For character code compares, two 7485 comparators are connected in series to perform the six-bit comparison. The type and code comparators are connected to the inputs of a 74153 multiplexor, which then generates the match output. If the loop bit (TT_l) is set, the match output is always asserted. Otherwise, the match output is selected from either the type or the code comparator based upon the value of the type/code bit TT_t. When a CM is in the idle state, the TT address will be 0. The contents of this TT word will have TT_t=1 and TT_c=0, which will

inhibit matching regardless of the input, since at least one type bit must be set to generate a match. In any other state, the match output is determined as explained above, and if a match is detected, the 74148 encoder causes the multiplexor (N) to load the address register (A) from the next-state address in TT_n. If a match occurred in a terminal state (one for which TT_h, the hit bit, was set), the state address and CM number in H is gated to the MC during Ta.

2.7.3 Comments on the MSI Term Matcher

Although every effort was made to build the MSI matcher in a manner that could be easily translated into an LSI version, several implementation considerations complicated this.

For an LSI implementation, the primary design consideration would be pinout limitations. Since 64-pin packages are not uncommon even at present, the fact that the MC requires 49 pins and the CM′s each require 47 pins in the MSI implementation presents no major problem. No particular effort was made to minimize output conections in the MSI version, because doing so would have complicated the design and illustrated nothing in particular of interest. However, it does not take much imagination to think of ways that pinouts could be reduced significantly.

First, a 16-bit bus is used to connect the MC to the query resolver. This width was chosen primarily because a PDP-11 is being used as the query resolver in the experimental system. This could easily be changed to 8 bits with little trouble (the main problem would be reading out of the FIFO 8 bits at a time). In the CM′s, the fork address is a full 7 bits. This allows a CM to fork to any address in a neighbor. A small portion of the address space is adequate to contain the fork states, and it does not seem unreasonable to restrict the addresses which can be forked to to the same range. Experience with practical sized state tables indicated that sixteen fork states was more than adequate for a CM, similarly sixteen states which can be forked to appears to be adequate. This would reduce by 9 the number of pins on each CM (to 38). Even the addresses of terminal states could be restricted to allow the 10-bit H

register to be cut to 8 bits, but this seems to be stretching things a bit.

Connecting the outputs of the memories directly as inputs to other devices makes it difficult to load them unless chips with separate data outputs and inputs are used. The only suitable such memory is the 2101-series, of which the MWS5101A is an example. Unfortuantely, the only quantum this chip is available in is the 256x4 bit size. This results in most of the memory in each CM being wasted. First of all, only 64 words of 13 bits are needed for the ST (four type bits are used in the prototype only because they are available). Thus, 3264 of the 4096 bits in the ST are unused. Second, only 128 words of 16 bits each are needed for the TT′s. The memory wasted is 2048 of the available 4096 bits. Having 128 states per CM is more than adequate even for 120-term tables (using the results of Chapter 3, we can compute the average number of states per CM as 73 for 876 state tables, corresponding to 120 7.3-state terms). Finally, an FT size of 16 words of 10 bits is all that is really needed, so 3936 of the 4096 bits of FT memory are wasted. In total, only 3040 of the total 12288 bits in the RAM chips in each CM are used. This is under 25% of the memory. Obviously, if the CM′s were built using LSI, the memories could be configured to exactly the size desired.

Another peculiarity of the 5101 memories is the fact that from the time output-enable is asserted until data becomes available is 150 nanoseconds. Rather than use a multiplexor to select the next state address, it would be simpler to connect the next state outputs of all memories to the inputs of the A-register. When the source of the next address was determined, the appropriate source could be output-enabled and then strobed into A. Unfortunately the enable time for the 5101′s is too slow for this; either separate tri-state buffers must be placed on the next-state bit′s data outputs or a multiplexor must be used. The latter choice required fewer packages.

Finally, in an LSI implementation, the sequencing and mode logic would be implemented using a PLA rather than flip-flops. PLA chips are available which could have been used in the MSI prototype, but

programmming one was not considered to be worth the effort, especially considering that no PLA programming hardware was available. Furthermore, any changes to the circuit would require re-programming the PLA.

2.8 LSI Implementation of the Matcher

Actual layout of the NFSA Term Matcher as one or more LSI chips was far too time consuming to attempt, especially since no computer design aids were available. However, the claim was made that the NFSA Matcher was amenable to LSI implementation, so some justification of this is in order. Several aspects of the design which simplify LSI implementation have already been mentioned. The circuitry contains relatively little random logic; it is primarily composed of memories and registers. Much of the logic necessary is easy to implement efficiently in NMOS. The comparators, the multiplexor, the encoders and decoders, the FIFO, and the MC′s state machine all can be mapped directly into NMOS structures described in [MeCon80]. The data paths are narrow; one internal bus is 16 lines wide and the others are eight or fewer. The timing is designed to be non-critical, and the gating scheme could easily be changed to work with a two-phase clock common to NMOS LSI circuits. Additionally, the operations are overlapped so that a slow memory can be used. For match cycle times in the 800 nsec. range, memory speeds of around 600 nsec. are adequate (allowing 200 nsec. for matching and selecting the next address). The amount of memory per CM is also not unreasonable (slightly under 3K bits). With 16K bit static RAM chips already available, putting one or more CM′s together on one chip is within the bounds even of current technology. In addition to these brief comments, several other areas will now be covered in more detail.

2.8.1 Packaging

A separate matcher is needed for each track being searched at a given instant, so for large databases presumably many searchers would be needed (perhaps as many as one per read head for each disk drive). Obviously implementing a matcher using as few IC packages as possible is important.

The obvious way to package the matcher would be to have one chip for the MC and another for the CM. The pinouts for the MSI MC and CM were illustrated in Figures 2.14 and 2.15, but as was discussed, LSI packaging constraints would force some modification to these.

A production system would need more than eight CM's, so the CM identification would need to be expanded. Sixteen CM's are adequate to handle tables of over 120 terms (as will be shown in Chapter 3), so a 4-bit CM ID is sufficient. This expands the number of bits in the K-register to eight, and requires (with the 7-bit final state address) an 11-bit H-register to report hits. Reducing the fork address to four bits was also discussed, and would save nine pins in each CM. Finally, internally generating the RESET signal in the MC and interfacing the MC and QR via an 8-bit bus reduce the number MC output pins.

The result of these interface changes is that the CM now requires 39 pins, and the MC requires 41. Standard IC packages with more pins are currently available, but it is worth expending some effort to be clever and eliminate one pin from the MC to allow using a standard 40 pin package. By encoding the MATCH line as an unused combination of the memory select code (all ones, for example) the K-register can be reduced again to seven bits. The CM would then require 38 pins and the MC, 40 pins.

While this packaging is an immense improvement over the MSI version, the prospect of building the largest possible matcher (one MC and sixteen CM's) using seventeen 40-pin packages is still not particularly appealing. Since each CM requires only 3000 bits and 16K-bit static RAMS have already been introduced, it should be possible to fit four CM's together on a chip with similar technology. This would reduce the sixteen-CM matcher to five packages. By restricting forking so that forks only are taken to CM's in the same chip (the four CM's in each chip being connected in a ring as in Figure 2.7) each CM package only requires 25 pins (power, ground, 3 clocks, 7 for the K and C registers, 11 for the H and D registers, and two for the high-order CM address). It is possible that restricting forks in this manner could prevent assigning table that would fit using the

standard interconnection scheme. Whether this method of packaging substantially decreases the size of the state tables that can be handled by a given number of CM's will be discussed in Chapter 3.

Only one problem remains, the necessity of designing, fabricating, and stocking two different chips (the MC and quad-CM). However, looking at the interface between the MC and the CM's suggests a solution. The MC's only output connections to the CM's are the three clock signals (Ta, Tb, and Tc) and the K-register. Consider a packaging scheme placing one MC and four CM's on one chip (Figure 2.19). Forks would be restricted to on-chip CM's as above, and the MC-quad-CM chip would have C-bus and H-bus pins just as the MC does. The MC would have an additional input which could force the clock signals and the C-bus to be output disabled. A disabled MC would be completely passive, having no effect on the operation of the one enabled MC and the rest of the CM's. By connecting more than one such chips together and only enabling one of the MC's, matchers of any desired size could be configured using one type of chip. Naturally, only the chip with the active MC would be connected to the disk and query resolver. Each chip would then require the same number of pins as the original MC chip (forty) plus two for the high order CM address and one to disable the MC. It is not clear how this would be reduced to below forty, but fortunately there are standard 48 and 64 pin packages which would work nicely.

A final alternative is to build the Term Matcher using CMOS. The MC and all sixteen CM's would be built on as many chips as necessary. The leadless chips would then be tested and packaged in a quasi-`hybrid' form on one substrate. The only I/O pins would be those needed to connect the matcher to the query resolver (12), the disk (5), and power (2) - a total of nineteen. In this manner, the entire Term Matcher could be fit into one 24-pin package! The low power consumption of CMOS makes heat dissipation no problem; in fact, this hybrid packaging technique was suggested for fitting a 16K-byte CMOS static RAM on a single 24-pin DIP in [Woll80].

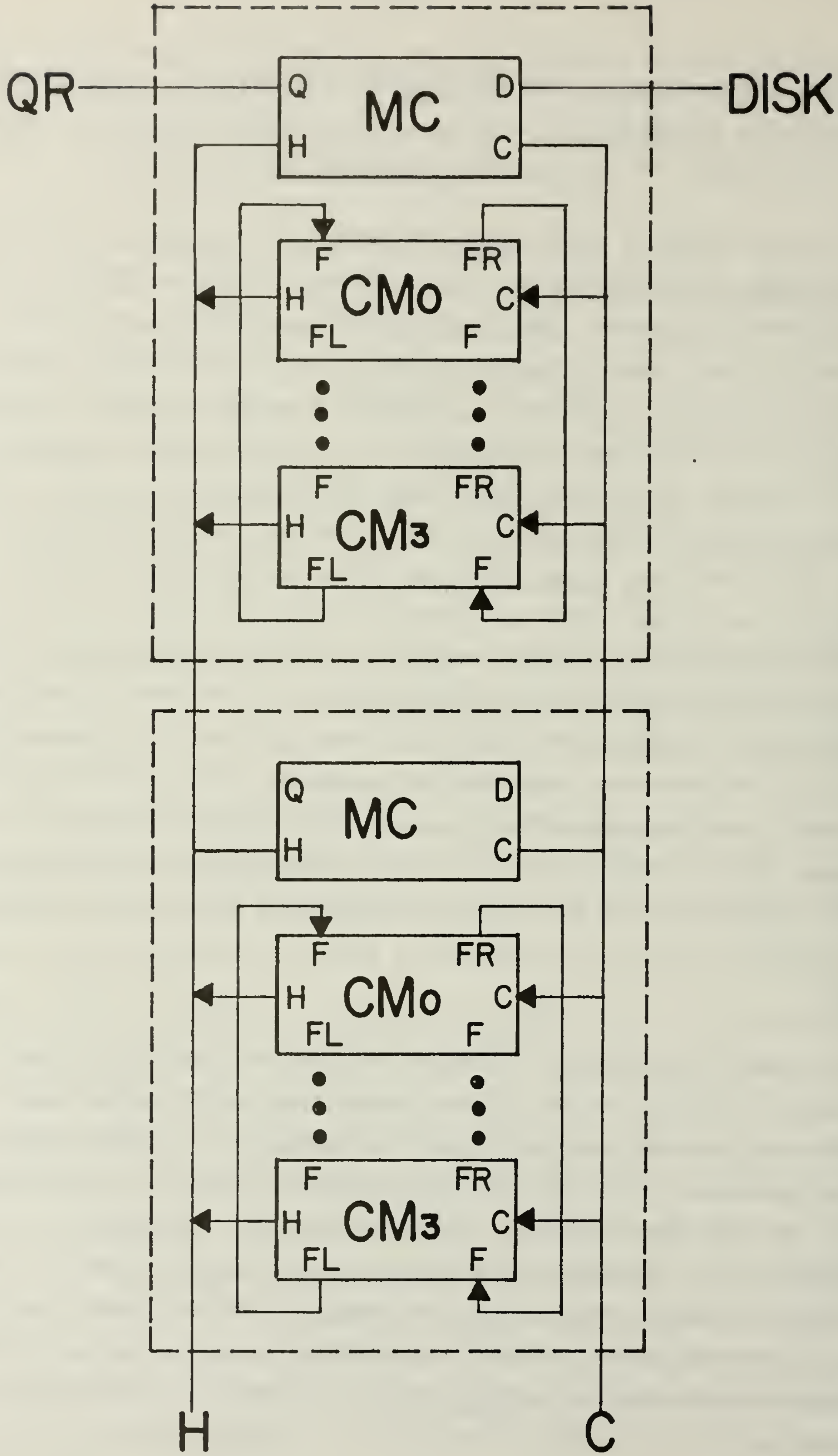


Figure 2.19 LSI Packaging Scheme for NFSA Term Matcher

2.8.2 Testing

In large integrated systems of the size of the NFSA Term Matcher, it is important to consider testability. Device testing is a problem that has not been solved even for circuits of the complexity typical today. It is usually not economically feasible to exhaustively test the function of every circuit even during manufacture, much less in the field. The details of how individual devices are tested are typically implementation dependent, and thus little can be said at this time about the specifics of how an LSI NFSA Term Matcher would be tested. However, the basic design of the NFSA lends itself to testing.

The obvious strategy for testing the NFSA is to load it with a known state table, input a known sequence of inputs, and monitor the operation to see if the correct sequence of outputs is generated. The matcher contains little combinational logic (a counter, a few buffers and registers, the next-state multiplexors, and the comparators). Relatively short test sequences would be sufficient to detect most faults in the data paths and logic.

Detecting faults in the memories would be more difficult; the sixteen-CM matcher has nearly 48K bits of memory. However, most memory faults are also easy to detect. For example, consider how the Transition Table (TT) might be tested. The test table would be designed so the diagnostic input sequence drove the CM through all states. If the character code field TT_c is wrong, the match fails and instead of continuing in the match sequence, the CM returns to the idle state. If the next state address is wrong, the CM will go to an unexpected state. In either case, if the output sequence is properly designed, a later output will not be what is expected. Faults in the type/code select bit TT_t cause matches to fail unexpectedly, and faults in the loop bit TT_l also are easy to detect: the CM either mistakenly stays in the state (stuck at 1) or does not (stuck at 0). Faults in the hit bit TT_h either cause spurious outputs or prevent an output from being generated when it should. Such techniques would not exhaustively test the device (for example, they do not necessarily detect pattern dependent failures) but

they would provide a fairly good confidence test. Additionally, by recording test sequence on one or more cylinders of the database disks, the matcher could be periodically tested while the system remained in operation.

If sufficient room was available on the chip, extra logic could be added to test the memory. The simplest alternative would be to allow the contents of the memories to be read back by the query resolver, although this would require that the memory outputs be multiplexed together and routed back along one of the busses. Not only would extra logic be needed to do this, but extra pins would be necessary to control the read-back function.

2.9 Summary

This chapter introduced a new model for text searching, the nondeterministic finite-state automaton. It discussed how the NFSA could be used to search for terms and other patterns in the environment of a text database system. Minor restrictions to the fully-general NFSA were imposed (concerning transition labels and fanout limitations), which simplified the task of implementing it in hardware. The advantages that the NFSA model has over the FSA and other models was also discussed. The NFSA offers advantages in terms of simplicity, flexibility, and implementation over alternative designs, and represents a significant advance in text searching methods.

Alternative approaches to the implementation of the NFSA Term Matcher were next discussed. First, a design using multiple, identical FSA's to interpret the state table by effectively `creating a new copy' of the automaton to pursue alternative search paths, was discussed. Problems with this approach involved the difficulty of dispatching free FSA's to follow new search paths and contention for access to the state table memory by multiple active FSA's.

A much better implementation was then introduced, in which the state table is partitioned among several interconnected sub-machines. Each sub-machine (called a character matcher, or CM) was only capable of occupying

states in its block of the partitioned table. The partitioning algorithm guaranteed that only one state in each block could be occupied at any time, by only assigning compatible states to each block. Each block of the state table resided in a memory dedicated to the sub-machine responsible for it, so memory contention was eliminated. The sub-machines had the capacity to start themselves in response to startup sequences in the input stream leading to states in their block of the table. The sub-machines were interconnected to allow a limited capability for one sub-machine to activate others to support certain types of transitions (forks). The self-starting and forking features eliminated the necessity of externally scheduling the sub-machines.

Actually building the NFSA matcher was then discussed. A prototype matcher was designed using MSI circuits, its purpose being to verify that no design problems were being ignored, and to serve as a testbed for performance measurments in an experimental system. Questions involving an LSI implementation were then discussed. It was of course unrealistic to actually design an LSI version of the NFSA Term Matcher (neither the time nor the facilities were available), but several facets of such a design were discussed. It was shown that by several yardsticks (total amount of logic, pinouts, and testability) an LSI version seems feasible even with present technology; of course the rapid development of semiconductor technology will make circuits of this complexity commonplace in the near future.

Finally, how the NFSA Term Matcher fulfills the design goals stated in Chapter 1 should be discussed. No memory buffers at all are necessary to hold text; the data is matched directly off the storage medium. Complex matching operations are supported; examples of variable-length don't cares (initial and embedded) were shown, as were type-matches and fern recognition. LSI implementation seems practical, at least no reason why this would be difficult has presented itself. The final two goals were configurability and minimization of channel bandwidth. The necessary bandwidth is that to load tables and report hits. Intuitively, this should be lower than that necessary to transfer the text to a centralized processor for searching. Configurability will also be discussed later,

but again intuitively the number of terms that the matcher will handle can be increased almost arbitrarily simply by adding CM's.

In summary, a model was introduced which supports a powerful set of search operations on text data. A machine has been designed to implement this model, in which a nondeterministic state table is paritioned among several identical processors. How these tables can be efficiently partitioned will be discussed in Chapter 3.

Chapter 3

State Assignment

The NFSA state table is partitioned and states are assigned to CM's before the search is started. In particular, before a region is searched, the system must determine all queries being searched for in the region. Then, all terms from the queries are collected and the corresponding state table is built. The table is loaded into the CM's, and the search is performed.

As Chapter 2 pointed out, the FSA scheduling logic and memory contention problems inherent in the straightforward multi-FSA implementation of the NFSA Matcher could be eliminated in trade for performing the extra computation to partition the state table. This is a good trade only if state tables can be partitioned quickly enough to keep up with query arrivals.

Assigning states to CM's is conceptually related to two familiar processes from sequential machine theory: machine decomposition ([HaSt66]) and state minimization of incompletely specified machines ([Booth68]). While this fact should cause encouragement by providing ready-made algorithms for paritioning state tables, in reality it does not. Actually, the partitioning problem can be stated in the following manner:

To partition an N-state table among k CM's, find k vectors a^k of N elements each, such that:

$$a^i \cdot a^j = 0 \text{ for } i \neq j \qquad (1)$$

$$\sum_{i=1}^{k} a^i = \overline{1} \qquad (2)$$

$$(P \times a^j) \cdot a^j = 0 \text{ for all } 1 \leq j \leq k \qquad (3)$$

where all operators (+,x) are boolean (i.e. `or' and `and' respectively).

P is an array such that $P_{ij}=1$ iff states i and j are incompatible. Vector a^i contains elements $a^i_j=1$ for each state j assigned to CM_i.

Equations (1) and (2) insure that each state is assigned to exactly one CM. Equation (3) insures that all states in each CM are compatible. The point of this is that finding solutions for a^k is NP-complete. This can be demonstrated by reducing (3) to the satisfiability problem. For example, if N=2,

$$\begin{bmatrix} P_{11} & P_{12} \\ P_{21} & P_{22} \end{bmatrix} \times \begin{bmatrix} a_1 \\ a_2 \end{bmatrix} \cdot \begin{bmatrix} a_1 \\ a_2 \end{bmatrix} = 0 \tag{3a}$$

Expanding and simplifying,

$$P_{11}a_1 + P_{12}a_1a_2 + P_{21}a_1a_2 + P_{22}a_2=0$$

Applying DeMorgan's Theorem and negating,

$$(\overline{P}_{11}+\overline{a}_1)\cdot(\overline{P}_{12}+\overline{a}_1+\overline{a}_2)\cdot(\overline{P}_{21}+\overline{a}_1+\overline{a}_2)\cdot(\overline{P}_{22}+\overline{a}_2)=1 \tag{3b}$$

The P_{ij} (pairwise incompatibilities) are constants for a given table, and their values (1 or 0) can be substituted into (3b) to simplify it. For example, suppose $P_{11}=P_{22}=1$, and $P_{12}=P_{21}=0$:

$$(\overline{a}_1+\overline{a}_2)\cdot(\overline{a}_2+\overline{a}_1)=1 \tag{3c}$$

Equation (3c) is in Conjunctive Normal Form, and thus is an instance of the satisfiability problem. Any instance of (3) can be similarly reduced, and since the reduction can be done in a polynomial number of steps on a deterministic processor, finding solutions to (3) is NP-complete. We can therefore expect the time needed to compute assignments to grow exponentially if any straightforward algorithm is used. Some of the state tables which must be partitioned contain nearly 1000 states, and on tables of this size the textbook algorithms are far too slow to run in real time.

Thus, as is often the case, the algorithms from the literature are impractical for direct implementation and are of use primarily as a point of departure in the development of more efficient algorithms. The state assignment algorithms can be speeded up due to two facts:

1. The number of CM's is fixed for a given matcher. Thus, in production, state assignment is a first-fit rather than a best-fit problem. Standard state minimization, on the other hand, is a best-fit procedure.

2. Certain characteristics of the match process allow one to state a priori that certain states can reside in the same CM, even without explicitly testing them for compatibility. For example, if two terms A and B are restricted to start on word boundaries and begin with different characters, all states in A are known to be compatible with all states in B. Taking such characteristics into consideration, methods can be developed to drastically reduce the number of compatibility tests.

3.1 Definitions

Before proceeding in depth with a discussion of state assignment techniques, it will be helpful to define some of the terms which will be used:

STATE - a condition in effect in a given CM at a particular time, defining what action will be taken in response to the receipt of the next input character.

TRANSITION - the action taken by a CM in response to receipt of an input character. This action is one of the following possibilities:

1. Change to a different state (normal matches & mismatches)
2. Change to a different state and force one or more neighboring CM's to specified states (fork state matches)
3. Remain in the same state (loop state mismatches)

4. Remain in the same state and force one or more neighboring CM's to specified states (loop state matches)

TRANSITION LABEL (or just LABEL) - defines what subset of the input character set will cause a given transition to be taken. To generalize the definition to describe transitions out of the idle state, the label consists of both a match specifier (either an explicit character code or a type specifier) and a previous character type specifier. The match specifier attribute is called the CHR, and the previous character type is called the PCT. For labels of transitions not originating in the idle state, the PCT is the type of the CHR attribute of the label on the transition leading into the previous state (Figure 3.1).

START STATE - any state entered via a transition out of the idle state. Usually, start states correspond to the first character of a term.

IDLE STATE - a state in the table out of which transitions are defined for the first character in each term and into which transitions are implicitly defined for all mismatches (except those occurring in loop states). The idle state can be thought of as always being occupied in the sense that receipt of a character sequence corresponding to the label of a transition out of the idle state will always be taken regardless of what other states the NFSA is occupying, although normally the state assignment will insure that transitions are not taken from the idle state into an active CM. In practice the idle state is split among all CM's; each CM's idle state contains outbound transitions to all start states residing in that CM.

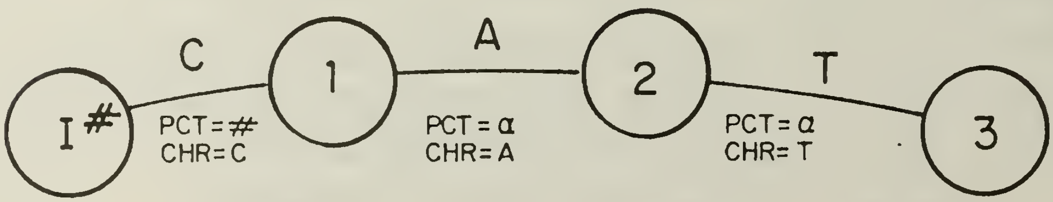


Figure 3.1 State Transition Labels

DISJOINT - two labels are disjoint if it is impossible for transitions marked with them to ever be taken in response to receipt of the same two-character sequence. More formally, two labels are disjoint if any of the following are true:

1. The two labels′ PCT attributes have no types in common.
2. Both labels′ CHR attributes are character codes (not types), and the codes are not identical.
3. Both CHR′s are type specifiers, and they have no types in common.
4. One CHR is a character code, the other is a type specifier, and the code is not of a type included in the type specifier.

The definition of <u>disjoint</u> can be extended to comparisons of single attributes of labels (such as comparing CHR of one label with PCT of another), in which case only Rules 2, 3, and 4 are applied. Obviously in this case the attributes being compared are both considered as CHR′s when applying the rules.

INTERSECTING - if two labels or fields are not disjoint, they are said to <u>intersect</u>.

LOOP-DISJOINT - Loop states have two inbound transitions (the one from the previous state and the loopback transition). Disjoint tests are used primarily to see if a given input can cause transitions into two states to be taken simultaneously, but if one is a loop state, the transition can come from the previous state or the loop state itself. Consider state S and loop state L in Figure 3.2 If L is occupied, the previous inputs must have been either \`xy′, \`yz′, or \`zz′. If S is occupied, the previous inputs must have been \`pq′. The transitions leading into L and S are <u>loop-disjoint</u> if the three former sequences are all disjoint from the latter. More formally, the transition are loop disjoint if <u>all</u> the following are true:

1. PCT_y and PCT_q are disjoint or CHR_y and CHR_q are disjoint.
2. CHR_y and PCT_q are disjoint or the loop label z and CHR_q are disjoint.

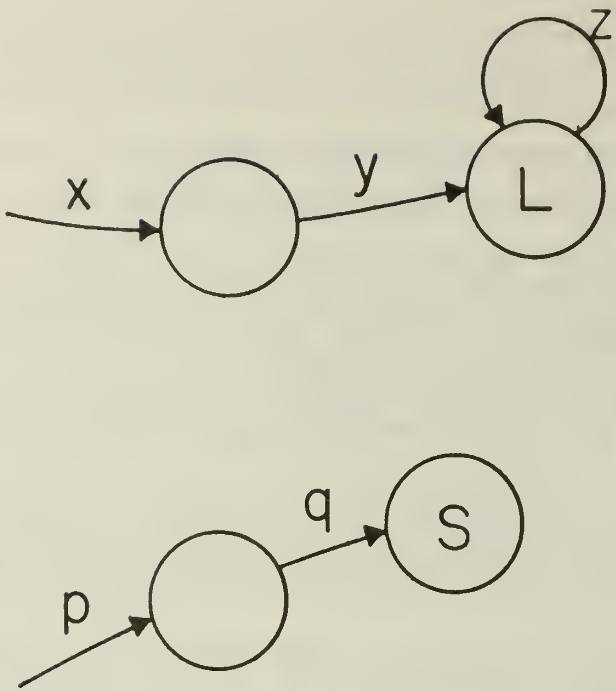


Figure 3.2 Loop Transition Labels

3. z and PCT_q are disjoint or z and CHR_q are disjoint.

COMPATIBLE - two states are compatible if it is impossible for them to be occupied simultaneously (exception: the idle state). This allows them to be assigned to the same CM. More formally, compatibility can be tested by applying the following rules in order until one is found to apply:

1. If either state is an idle state, then the two are compatible.
2. If one is a loop state, the other is a start state, and if the two are not loop disjoint, then the two are incompatible.
3. If one is a loop state, the other is not a start state, and if the loop state either is or is incompatible with the other's previous state, then the two states are incompatible.
4. If the incoming transitions of the two states are disjoint, then the states are compatible.

5. If either is a start state, then the two are incompatible (due to Rule 4, the inbound transition labels intersect).
6. If the previous states of both are either the same state or are incompatible, then the two are incompatible.
7. If none of the above apply, then the states are compatible.

3.2 Best-Fit (Minimum CM) Assignment

Although in production state assignment is a first-fit problem because of the fixed number of CM′s per matcher, it is necessary during the system design to decide how many CM′s to build into each matcher. How this decision is made for an individual system will be discussed later, but basically it is made by first deciding a size distribution of the number of terms in state tables likely to occur in production operation of the system. A best-fit assignment program is then executed using random sets of search terms of these sizes and computes the minimum number of CM′s needed to match each table. The matcher will be built with a number of CM′s adequate to handle all but some small fraction of the random test tables. During production operation, tables that are too large to fit in the available number of CM′s are divided up and searched in multiple revolutions. Although the table size distribution will vary from system to system, it is anticipated that it will be necessary to handle tables having in the neighborhood of 1000 states (about 120 terms). Thus, assignment techniques will be evaluated using a 1000-state upper bound.

While it is not necessary to compute best-fit assignments in production operation of the system, it is still necessary to compute them for the test tables. However, for 1000-state tables, the textbook algorithms are so inefficient that they can not even be used for this purpose. This section will illustrate this by reviewing pertinent topics from automata theory. It then will introduce modifications to these techniques and outline a procedure for computing nearly-optimal assignments in reasonable amounts of time and memory. Finally, results will be presented showing how the number of CM′s needed varies with the state table size.

3.2.1 Machine Decomposition

The technique of machine decomposition ([HaSt66]) offers a method for breaking a large machine into smaller component machines with well-defined interconnections. Given the state table defining a machine M, it allows describing M in terms of interconnected sub-machines M_i. In this case, M would correspond to a deterministic machine derived from the NFSA state table, and M_i would correspond to the CM's necessary to match the table. States in each of the component sub-machines are derived from partitions of the state set of M:

$$P=\{B_i\},\ B_i \subseteq \{Y\}$$

$$B_i, B_j \varepsilon P \text{ and } i \neq j \Rightarrow B_i \cap B_j = \varphi$$

$$\cup_{i=1}^{k} B_i = \{Y\}$$

where {Y} is the state set of M, and p is a partition composed of blocks B_i, each block containing states from {Y}, such that each state appears in exactly one block. The component machines M_i can be described in terms of `partition pairs':

$$(p_i, q_i) \mid p_i \times \{X\} \rightarrow q_i$$

where {X} is the input alphabet of M. In simple terms, M_i is defined by a partition pair (p_i, q_i) such that blocks of the preserved partition q_i define the states of M_i and p_i contains the information needed to compute the next state of M_i for any given input.

Finding a minimal assignment of states to CM's is analogous to finding a minimal feedback decomposition of a machine M [Booth68]. From the set of partition pairs, it is necessary to select the smallest set M_i such that the g.l.b. of all q_i is a partition having one state per block (i.e. M's state can be decoded from the states of the M_i), with there being some ordering of the partition pairs such that $q_{i-1} * q_i * q_{i+1} < p_i$ (to satisfy the requirement that submachines (CM's) are only connected to

their neighbors), and with all blocks q_i corresponding to states in the nondeterministic table.

Computing assignments for NFSA tables of 1000 states by this method is, of course, quite impractical. To begin with, converting the k-state nondeterministic table into a deterministic one results in a table size of $k'=2^k$ ([HopUl69]). Second, the set of partition pairs must be generated. There are $O(k'(k'-1)/2)$ of these. Even if it were practical to compute these, finding an ordered set satisfying the connectivity and correspondence constraints would in general require checking all combinations of partition pairs. Machine decomposition is useful as an abstraction for viewing interconnected machines and describing their behavior, but it provides us with no efficient strategy for assigning large tables.

3.2.2 State Minimization

State assignment in the NFSA has an analog in the problem of state minimization for incompletely specified sequential machines, although the definition of compatibility differs between the two. In the latter, if for any input sequence two states of a machine M can never cause M to generate different outputs in a case where both outputs are specified, the states are compatible and may be grouped together in the same state of a minimal machine M′. In the former case, states are compatible if they can never be simultaneously occupied, in which case they can be assigned to the same CM.

Sources such as [Unger69] and [FrMe75] give algorithms for state minimization. These algorithms differ in detail, but in general consist of these steps:

1. Generate a table expressing compatibility between each pair of states (pair table)
2. Use this compatibility information to compute the set of `maximal compatibles′ (MC′s) - sets of compatible states which are not subsets of other compatible sets.

3. From the set of MC′s, find the set of `prime compatibles′ (PC′s). A PC is a subset of (the states in) an MC which is not dominated by another such subset ([FrMe75]).
4. Find a minimal closed set of PC′s covering every state of M such that elements of the class set implied by each PC under all inputs are contained in other PC′s in the set.

The states in each compatible in the cover are grouped together into one state of the minimal realization M′. Similarly, in an NFSA table assignment, the states in each compatible can be assigned to the CM associated with that compatible.

The reason for checking the class sets of each PC is to insure that the transitions out of all states in each PC_i in the cover M′ under a given input go to the same compatible PC_j - just as a state of M takes a transition to only one other state under a given input. The restriction on which compatibles are eligible for inclusion in the cover is slightly different for NFSA assignment. In the NFSA, states retain their identity, and all states in a compatible in the cover need not go to the same next state under a given input. Rather, the interconnection structure of the CM′s imposes a different restriction on the choice of compatibles for the cover: since CM′s can only be connected to two neighbors, there must exist an ordering of the compatibles in the cover M′ such that the transitions out of each state in a compatible under any input go only to states in the same or adjacent compatibles.

As was true for machine decomposition, direct application of the textbook algorithm is too inefficient to use on large tables even during the design phase. Computing the pairwise compatibles requires $N^2/2$ compatibility tests, and computing the maximal compatibles requires in the worst case $O(2^N)$ operations (these algorithms usually involve traversing part of a tree whose maximum depth is the number of states in the table). Choosing a minimal cover in general involves examination of all combinations of PC′s.

The following sections will discuss several aspects of determining the minimum-CM assignment of a state table, and will show how the above

mentioned algorithms were modified to allow these assignments to be computed quickly.

3.2.3 Lower Bound on Number of CM's - Unlimited Interconnection

Each CM in the NFSA Term Matcher is connected to two neighbors. This limited interconnectivity allowed branches in the search path to be pursued in parallel without the need for any centralized dispatching logic, while still providing a resonable limit to the number of output pins on each CM. However, it is an interesting question whether allowing unlimited interconnection would permit a smaller number of CM's to be used.

If unlimited interconnection is allowed, the number of CM's needed is equal to the largest number of simultaneously active states possible in the state table. Since a state would be capable of making a transition to a successor in any CM, the only restriction on assignment would be that no two incompatible states could reside in the same CM. The number of CM's required would be the number of states in the largest `maximal incompatible' - that is, the largest group of states such that each state in the group is pairwise incompatible with all other states in the group.

Recall that the most difficult part of finding a minimal state assignment was finding an ordered set of compatibles such that transitions went only to adjacent compatibles. This required an exhaustive search of the MC's. However, since the <u>number</u> of CM's is the item of interest, it is not necessary to find such a set - we only need determine its existence. When the maximal incompatibles (MI's) have been computed and the largest found, the procedure is done. MI's are computed in the same manner as MC's. Several good algorithms ([SinSh72],[Stoff74]) exist for doing this. Also note the advantage of computing maximal incompatibles instead of maximal compatibles. For example, Stoffers' algorithm requires $O(2^k)$ operations, where k is the number of states in the largest maximal group (compatible or incompatible, depending upon which is being computed). In the NFSA table for a list of search terms, every state in the table is compatible with almost every other state. Thus, when computing MC's, k is usually almost N (the number of states in the table),

and 2^k is very large. On the other hand (as will be shown later) k is usually less than 10 for most 1000-state tables when computing maximal incompatibles.

However, even computing the maximal incompatibles of large tables requires much computation and a large amount of memory. For example, for an N-state table, Stoffers′ algorithm requires two N-bit vectors per state to store the compatibility and identification fields; two megabits are required for N=1000. A program was written to compute pairwise compatibles as part of the general state assignment program to be described later. This program is capable of determining pairwise compatibility in about 400 microseconds (115 PDP-11 machine instructions). At this rate, the pairwise compatibility table takes three minutes to build. While these numbers show that the computation is no longer impractical, the algorithm can be optimized substantially. More importantly, these optimizations are directly applicable to the first-fit assignment problem.

3.2.3.1 Incompatibility Covers

The first step in computing the maximal incompatibles for an N-state table is building the pair table, which requires $N^2/2$ compatibility tests, and at least that many bits of storage. One way to speed up compatibility testing is to break the state table down into small portions and process them independently. The big N^2 process can be broken down into many smaller n^2 processes, where $n \ll N$. This can be done by subdividing the table into m blocks of n states each such that all states in each MI reside together in at least one block. The set of all such blocks will form a cover of all incompatible states in the table. Assuming that $(mn) \approx N$, the time to process all blocks would be around $mn^2/2$. The time to compute the original pair table was $N^2/2=(mn)^2/2$, so the speedup is around $N^2/n^2m = m$. In addition to the speedup, another benefit is that it is no longer necessary to store the entire pair table. Memory sufficient to store one n-state pair table is all that is required.

The problem, of course, is subdividing the table without resorting to extensive compatibility testing (which is just what we are trying to

avoid). Compatibility tests are recursive in nature and can be quite slow. Fortunately, methods exist for subdividing the table without performing any compatibility tests.

One method of building the incompatibility cover is suggested by the observation that states seem to be compatible with states having disjoint labels on their incoming transitions. Usually, the set of disjoint labels would end up being the letters of the alphabet, so the table would be divided into 26 blocks. If the table were divided into one block for each input character code, and states were assigned to each block whose character code intersected its incoming transition, then presumably it would be insured that all mutually incompatible states would reside in the same block. Unfortunately, an EVLDC loop state can be incompatible with states with other labels (its successors, for example). The problem can be remedied by including each loop state in all blocks.

Once the cover blocks are obtained, all states in each block must be tested pairwise to determine incompatibilities. This points out the real problem with this method of dividing the table. Testing the compatibility of two states with intersecting incoming transition labels also requires testing the compatibility of their previous states. This results in two tests being necessary to determine the compatibility of each pair of states. If the labels are restricted to alphabetic characters and the blocks are of similar size, the number of compatibility tests will be roughly $2\times26\times[(N/26)^2/2]$, which is 38462 for a 1000-state table.

A better way to define incompatibility cover blocks is by grouping terms with intersecting startup transition labels. For example, if a state is in a term starting with `#S', it can only (with two exceptions) be incompatible with states in other terms having intersecting startup transition labels (e.g. `#S', `#α', `#*', `*S', `*α', or `**'). The exceptions are startup states in IVLDC terms (`*OLOGY#') and continuous word phrases (such as `#EAT#MORE#POSSUM#'). IVLDC startup states are incompatible with all other states with intersecting incoming transition labels - the `*O' in `*OLOGY#' is incompatible with both `O' states in `#EAT#MORE#POSSUM#'. In continuous word phrases (CWP's), the state after

each word break is incompatible with all startup states with intersecting inbound transition labels - the `P' in `#EAT#MORE#POSSUM#' is incompatible with the `P' in `#PUNK#ROCK#'. Actually, these are both instances of the same phenomenon - a startup transition with a label intersecting that of an internal transition of another term.

It is not difficult to divide the table into blocks containing terms with disjoint startup transitions. Each block in the cover will have a characteristic label which is the greatest lower bound of the startup labels of the terms in the block. This cover can be formed as follows:

Algorithm INCOVR (Build Incompatibility Cover Blocks):

1. Create a block for each unique startup transition. The label of this block is the label of the transition.
2. Test each pair of blocks to determine whether the characteristic label of one is a subset of the other (i.e. every character sequence matched by the subset is matched by the superset). If so, delete the block with the label that is the superset of the other.
3. Continue pairwise testing and deletion until no further deletions are possible.
4. Assign to the block all terms containing a startup transition label intersecting the block label.
5. Assign to the block all terms containing internal labels intersecting that of a startup transition of a term assigned to the block by Rule 4.
6. If the block contains a term whose startup transition is not loop disjoint with the incoming transitions of some loop state, assign the term containing the loop state to the block.
7. If a term is assigned to block B_i on the basis of its startup label, and is also assigned to block B_j on the basis of an internal transition label, then assign all terms in B_i containing a loop state to B_j.

Normally there will be a characteristic label (and hence a block) for each character of the alphabet. All terms with startups intersecting the

characteristic label will be assigned to the block; no other startup states could be part of a maximal incompatible containing all startup states in the block (they would be incompatible with at least one of them). However, all maximal incompatibles will still be generated since terms are assigned to all blocks whose characteristic intersects their startup label.

Rule 6 corresponds to Rule 2 of the compatibility test (Section 3.1). Compatibility is not tested per se during block building (avoiding such tests is the point of building the blocks in the first place). This rule insures that incompatible loop states will be included in a block without prematurely doing the much slower comptibility test.

Rule 7 may not be obvious. Consider a non-start state from B_i included in B_j as stated above. The only states it can be incompatible with are others with intersecting labels (which will also be included in B_j) and loop states in B_i. Automatically including terms from B_i with loop states in B_j avoids any compatibility testing during block construction, and still insures all states in each maximal incompatible will be in the same block.

Once the blocks have been built, the maximal incompatibles in each can be found. INCOVR is an improvement over the original blocking method in that no compatibility tests between states in different blocks are done and it is only necessary to store the pair table for one block at a time. No redundant compatibility tests are necessary even when testing a pair of states requires also testing their predecessors. If pairs are tested more than once, the result is quickly obtained from the table rather than being recomputed. Again assuming a similar size block for each character of the alphabet, the total number of pairwise tests necessary for an N-state table should be about $26\times[(N/26)^2/2]$ (or $N^2/52$), which is 19231 for N=1000.

The main disadvantage of this method is that the blocks can grow quite large. If the table contains an IVLDC term such as `*SEARCH', all terms containing an internal `S' will be included in one block. This will probably be the majority of terms in the table. Also, if the `*S' is

assigned to more than one block, each of these would contain the other `S′ states, causing compatibility tests to be repeated. It is possible, however, to modify INCOVR to rectify this. Instead of moving all the `S′ states into the block with the `*S′ startup, the IVLDC term is placed into all other blocks containing `S′ states. Instead of the block with the IVLDC getting very large, all the other blocks become slightly bigger. The above table blocking method can be modified to work in this manner as follows:

Modified Algorithm INCOVR (Improved Incompatibility Cover Builder):

1. Create the set of blocks as described in Rules 1, 2, and 3 above.
2. All terms having startup transition labels intersecting the block label are assigned to the block.
3. All IVLDC terms are assigned to any blocks containing transitions intersecting the IVLDC startup transition or containing loop states whose CHR intersects the IVLDC′s startup PCT.
4. All CWP terms containing internal transitions intersecting the startup label of any term in a block are assigned to that block.

Rules 3 and 4 require distinguishing IVLDC′s from CWP′s. This can be done, but the details are implementation dependent and it serves no purpose to discuss them in depth.

3.2.3.2 Term Tail Removal

Subdividing the state table using incompatibility covers allowed decreasing the storage and time necessary to compute pairwise incompatibilities by a factor of around 26. The idea behind the method was that each pariwise incompatibility must be found, but as few tests as possible should be done between compatible states. Testing states only against others in the same block resulted in a large reduction in the total number of tests. Even so, each state was tested against all other states in the block to build the block′s pair table.

The number of compatibility tests can be reduced still further by noting that even within a block, most states are compatible with all other

states. For example, in most cases if two states in different terms are found to be compatible, all subsequent states in both terms are also mutually compatible and need not even be checked. Any states which are compatible with all other states in the block need not be included in the pair table. Indeed, in a 1000-state table, only a small fraction of the states are incompatible with anything at all, and this fraction goes down for smaller tables. These states are typically clustered near the beginning of the terms containing them (in fact, most are start states). By cleverly selecting the order in which states are tested against one another, it is possible to find all incompatibilities while testing only a very small number of compatible pairs.

To minimize the number of compatibles tested, states should be checked term by term. Each term must be checked against itself (it may contain internal incompatibilities such as forks or loops) and against each other term in the block. Two terms are checked by applying the following algorithm to their startup states:

Algorithm DETAIL (Term Tail Removal):

1. If the two are the same state (when the term is being tested against itself) and some downstream (subsequent) state is a fork or loop state, the algorithm is applied recursively to test the successor against itself.
2. If the two states are incompatible, the algorithm is applied to their successor states (if both exist).
3. If one state is a startup state, and its PCT is intersecting the PCT of one of the other state′s successors, then the algorithm is applied to the start state and the nearest such successor. If both are start states, this rule is applied both ways. This rule catches incompatibilities internal to terms due to IVLDC′s and CWP′s.
4. If one state is a loop state and is incompatible with the other state, the algorithm is applied to the loop state and the other′s successor. Again, if both are loop states, this rule is applied both ways. This rule finds all incompatibilities due to

compatibility rule 3.

5. If either or both states are first states in tines of (possibly different) forks, repeat the algorithm to test all combinations of tines of the two forks.

DETAIL basically stops testing two terms as soon as it determines that they contain no more pairwise incompatibles. It in effect removes the 'tails' of terms from consideration. An explicit formula for the speedup afforded by this optimization is difficult to arrive at, since its effectiveness is determined by the location of incompatible states in the terms. However, experiments performed on sample tables (to be discussed in detail later) show that the speedup is dramatic. In fact, using tail removal to eliminate compatible states from consideration allows the maximal incompatibles to be computed with close to the theoretical minimum number of tests.

3.2.4 Computing Maximal Incompatibles

A program was written to choose terms at random from the index of a database (the one used is the Brown Corpus, a 1-million word collection of representative samples of English text) and compute the minimum number of CM's necessary to match the resulting state table. The program was written in PDP-11 assembler language (the only translator available on the machine having the Brown Corpus online). It was parameterized to allow various combinations of the above optimizations to be tried to gauge their effectiveness. This section will outline the program and present some of the results it obtained.

3.2.4.1 MAXINC Program Overview

The program to compute maximal incompatibles, MAXINC, is capable of building state tables of varying size, finding the minimum number of CM's necessary to match a table, and producing statistics about the computation. The operator enters the desired table size and the number of tables of this size he wishes generated. A facility is available to enable the operator to enter special terms (such as IVLDC's, CWP's, and EVLDC's). MAXINC then proceeds to generate the requested number of

tables. For each, the specified number of terms is chosen at random from the database index (a pseudo-random number generator is used to allow results to be repeated). The term text is then converted into a state table, and the table is processed to detect common prefixes and collapse them together into fork terms. INCOVR is then called to build the incompatibility cover, and then the pair table is generated, either by testing all pairs in the block or using DETAIL to generate the pair table without testing term tails. Finally, the maximal incompatibles are produced using a modification of Stoffers' algorithm. Each maximal incompatible is compared against the current largest, and the one containing the most states is saved.

When all blocks have been processed, statistics are printed regarding table size, compatibility testing statistics, and execution times for various steps in the program. The necessary number of CM's is also output. Optionally, other information can be printed, including the list of terms in the table, the state table itself, the list of blocks in the incompatibility cover, the incompatibles in each block, and the pair charts.

Figure 3.3 shows the sample output from one small table. The list of terms is at the top, followed by the list of cover blocks. Each block consists of several start state addresses followed by an octal -2. The list is terminated with a -1. Following the cover block list is the incompatible states and pair table for each block containing incompatible states. The fields in each state table entry are shown in Figure 3.4 The CM and CMA fields are used only by the first-fit assignment program, and are not filled in when computing maximal incompatibles.

Following each list of incompatible states is the pair table for the associated block. The first line (row) shows the bit vector (printed in octal) indicating the incompatible states of the topmost state table entry printed above it. Bit 0 (the low order bit) is set if the state corresponding to the row is incompatible with the first state in the list, bit 1 is set for the second, etc. Each state has the bit set indicating it is incompatible with itself, as required by Stoffers' algorithm. The

```
BRETHREN            BRAIN                 SOUTHEASTERN          SUMMERS
UNCIVIL             WORLD

470570
177776
471110
177776
471570
177776
471750
777776
177777

  ADDR     PCT CHR   PRV    NXT    FRK    CCH   CM CMA F  QTY INDX
  071010:  01  R    070570 071030 000000 070610 77 000 00 01  0000
  070610:  01  R    070570 070630 071010 000000 77 000 00 01  0004

00000000003
00000000003

00000000003

  ADDR     PCT CHR   PRV    NXT    FRK    CCH   CM CMA F  QTY INDX
  071410:  14  S    000000 071430 000000 071110 77 000 10 01  0000
  071110:  14  S    000000 071130 071410 000000 77 000 10 01  0004

00000000003
00000000003

00000000003

TOTAL # OF TERMS = 00006, TOTAL # OF CHARACTERS = 00044
00000 UNIQUE 1ST CHARS, 00000 FORK STATES, AND 00000 STATES IN TABLE
00002 COMPATIBLES FOUND, 00002 TESTED; 00002 INCOMPATIBLES FOUND, 00002 TESTED.
00000 COMPATIBILITY CHECKS, 00000 ASSIGNMENTS, # CM'S NEEDED IS 00002.

TIMES: BLDFK-00000  BLDCVR-00000  ASBLKS-00000
```

Figure 3.3 Output from Sample Run of MAXINC

ADDR - The PDP-11 memory address of the state table entry
PCT - The previous character type restriction attached to the label of the transition into the state. The bit definitions are:
Bit 0 (low order) Alphabetic
Bit 1 - Numeric
Bit 2 - Word Separator
Bit 3 - Fern (Context Separator)
CHR - Incoming transition label's match character (or type). The NFSA must receive this character to enter this state. If CHR is a type, its format is identical to the PCT.
PRV - Pointer to the entry of the previous state in the term.
NXT - Pointer to the entry of the next state in the term.
FRK - If the previous state forked to more than one state, the FRK field points to the next tine of the fork.
CCH - Field for the compatibility chain. Each state in the pair table is linked into this chain.
CM - Contains the CM number to which the state has been assigned.
CMA - Contains the address in the CM containing the state table entry.
F - Contains flag bits as follows:
Bit 2 - Loop state.
Bit 3 - Start state.
Bit 4 - Signals that some subsequent state is a loop state.
Bit 5 - Signals that some subsequent state is a fork state.
QTY - Contains the bitwise OR of the type of the CHR field in this and all subsequent states.
INDX - contains the index in the pair table of the entry corresponding to the incompatibles of this state. This is in multiples of the incompatibility bit vector size in bytes.

Figure 3.4 MAXINC State Table Entry

pair table is stored as a square bit matrix rather than a triangular one to facilitate the bit vector operations among rows done by Stoffers' algorithm. Following the pair table is the list of maximal incompatibles for the block. Their format is identical to that of rows in the pair table - a bit is set for each state in the incompatible and the bit position from the low order end corresponds to the row in which the state appears in the pair table.

At the bottom of the figure, statistics for the run are printed. The meaning of most of these is obvious, although several of the entries which get filled in during address assignment are 0. The number of compatibiles and incompatibles tested and found is a measure of the effectiveness of

INCOVR and DETAIL. Note that these numbers are low relative to the $N^2/2$ tests necessary if these optimizations were not used. The times printed at the bottom are in 100 microsecond increments, and represent the execution times of various parts of the program. BLDFK is the time necessary to detect forks in the nonreduced state table; BLDCVR is the time needed to build the cover blocks, and ASBLKS is the time needed to do the compatibility checking and find maximal incompatibles in all blocks. These are all zero because times are not recorded when debugging output (cover blocks, pair tables, etc.) is being printed.

3.2.4.2 MAXINC Results

Figures 3.5-3.7 show results obtained from runs of MAXINC. Each data point on the graphs represent the average obtained from runs on five tables containing the indicated number of randomly chosen terms.

Figure 3.5 presents statistics for the test tables generated by MAXINC. Curve 1 plots the total number of characters in all terms vs. the number of terms. It shows that each term has an average of 7.69 characters. Curve 2 shows the average number of states vs. the number of terms. Eliminating forks results in reducing the number of states by 5%. Curve 3 shows the number of incompatible pairs in the table. There are very few incompatible pairs for the small tables because in many cases there is only one term in some blocks, in which case no states in the block have incompatibles. For larger tables, all blocks contain more than one term and therefore have states with incompatibles. The average number of fork states in the table is shown by Curve 4. One might rightly complain that this curve should be plotted on a different scale, since it is so small relative to the other curves. However, that the number of forks is relatively small is just what the curve is intended to illustrate. The highest fraction of fork states to total states in the table occurs at 120 terms, where slightly under 4% of the states are forks. Reserving 1/8 of the state table for fork states (as was done in Chapter 2) is more than sufficient.

The results shown in Figure 3.6 illustrate how the addition of INCOVR DETAIL affect the total number of compatibility tests necessary to find

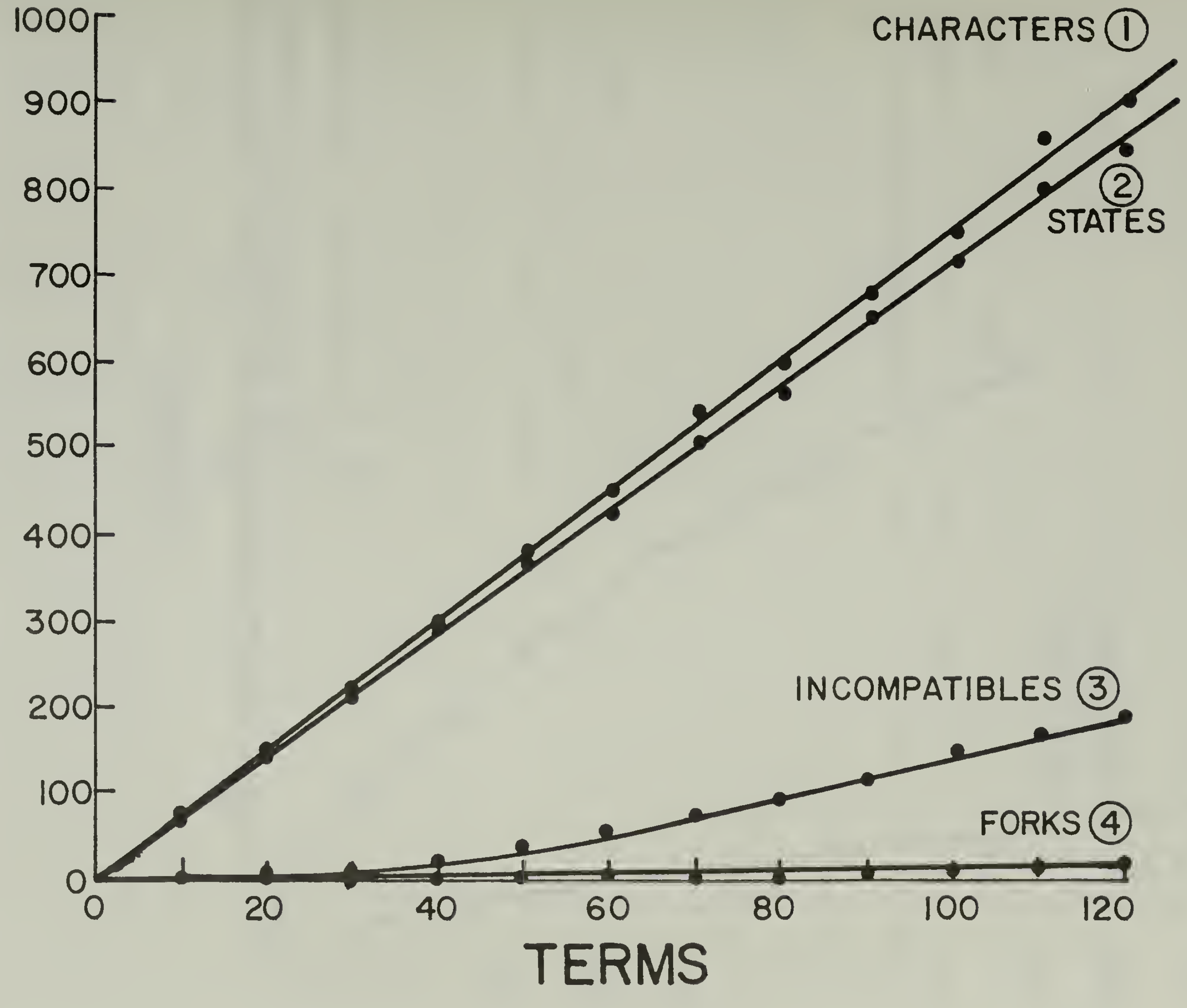


Figure 3.5 State Table Statistics

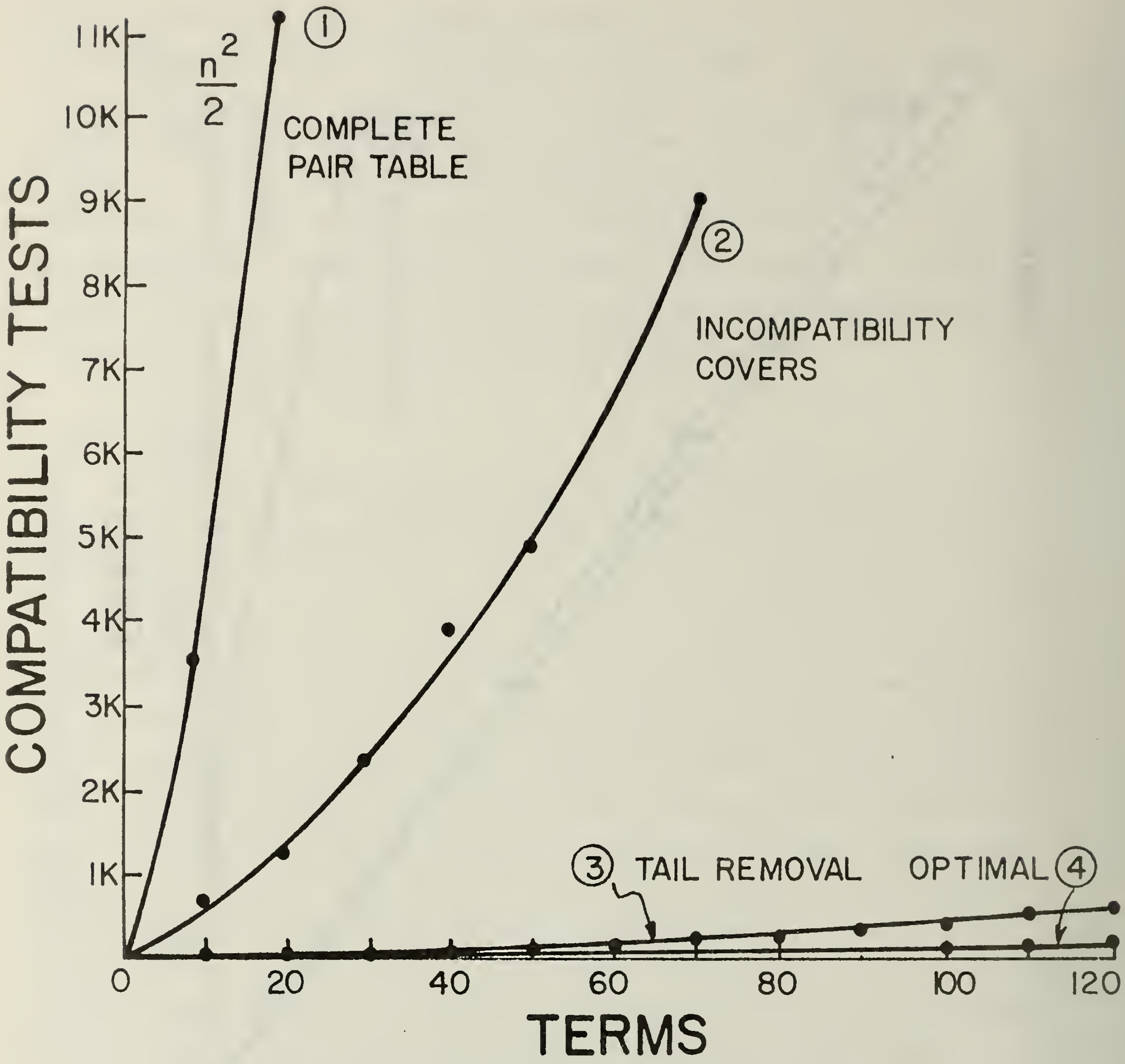


Figure 3.6 Compatibility Tests vs. Table Size

the maximal incompatibles. Curve 1 shows the number of tests to build the entire pair table for an N-state table ($N^2/2$). The number of tests needed when INCOVR is used to subdivide the table into smaller blocks is shown in Curve 2. This nearly twice the $N^2/52$ estimated in Section 3.2.3.1, mostly because several of the blocks are larger than average (e.g. terms starting with `S′) and some are quite small (`X′), so the total number of tests is

larger ($a>b>c$ and $a+c=2b \Rightarrow a^2+c^2>2b^2$). Still, Curve 2 is a great improvement over Curve 1.

Curve 3 shows the number of tests when using both INCOVR and DETAIL. The improvement over Curves 1 and 2 is obvious, but comparing Curve 3 with Curve 4 shows just how good the result is. Curve 4 plots the average number of incompatible pairs in tables of each size, and as such represents the minimum necessary number of compatibility tests. Curve 3 is very close to optimal over the range of table sizes being considered.

The actual time taken by MAXINC to find the maximal incompatibles is shown in Figure 3.7 Each data point represents the average time (BLDFK+BLDCVR+ASBLKS) needed to generate all maximal incompatibles for a table and select the largest. Curve 1 plots the times for MAXINC to generate tables using incompatibility covers alone, and Curve 2 plots the execution times using both incompatibility covers and term tail removal. The rise of Curve 2 is greater than linear, but it still rises quite slowly within the range of reasonable table sizes.

One may wonder why such attention is being paid to performance statistics for best-fit assignment, since first-fit will be used in production. The reason is that most of the time necessary in both best-fit and first-fit is that consumed in computing pairwise incompatibilities, and this needs to be done for both. INCOVR and DETAIL will speed up first-fit assignment as much as they did MAXINC.

Figure 3.8 shows the statistic which this section originally set out to determine. For tables of from 10 to 120 terms chosen at random, MAXINC found the number of CM's required. Each number on the field of the graph corresponds to how many test tables containing the number of terms shown on the X-axis required the number of CM's to match them shown on the Y-axis. Twenty tables of each size were generated, for a total of 240 tables. For a table of a given size, the number of CM's required falls into a narrow band. Only 3 of the 240 tables fall above the line $CM=5+(T/16)$, where CM is the number of CM's required and T is the number of terms in the table. The graph also shows that large tables of 120 terms (approximately 1000 states) will fit easily into a 16-CM system.

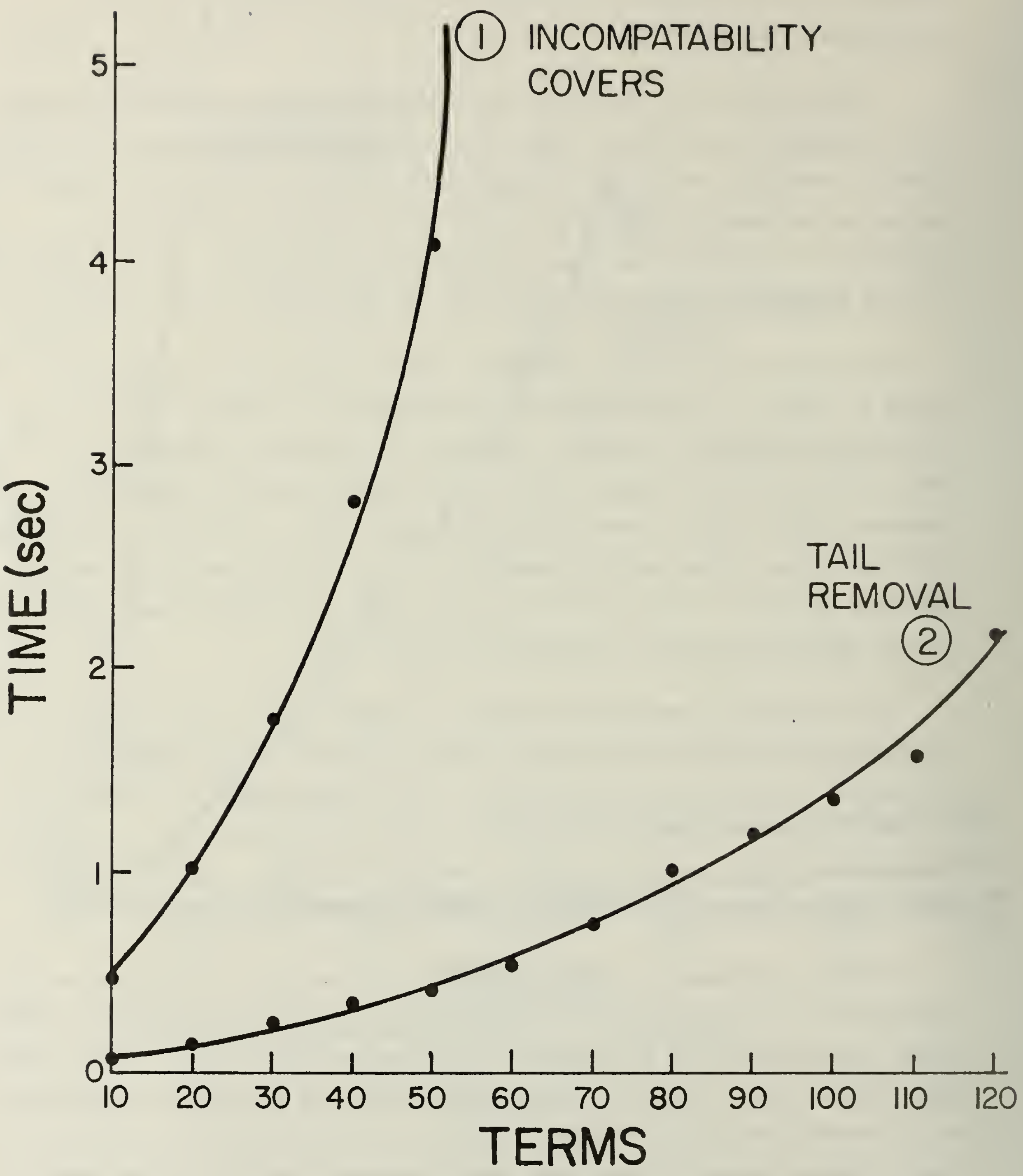


Figure 3.7 Number of Terms vs. Time for MAXINC

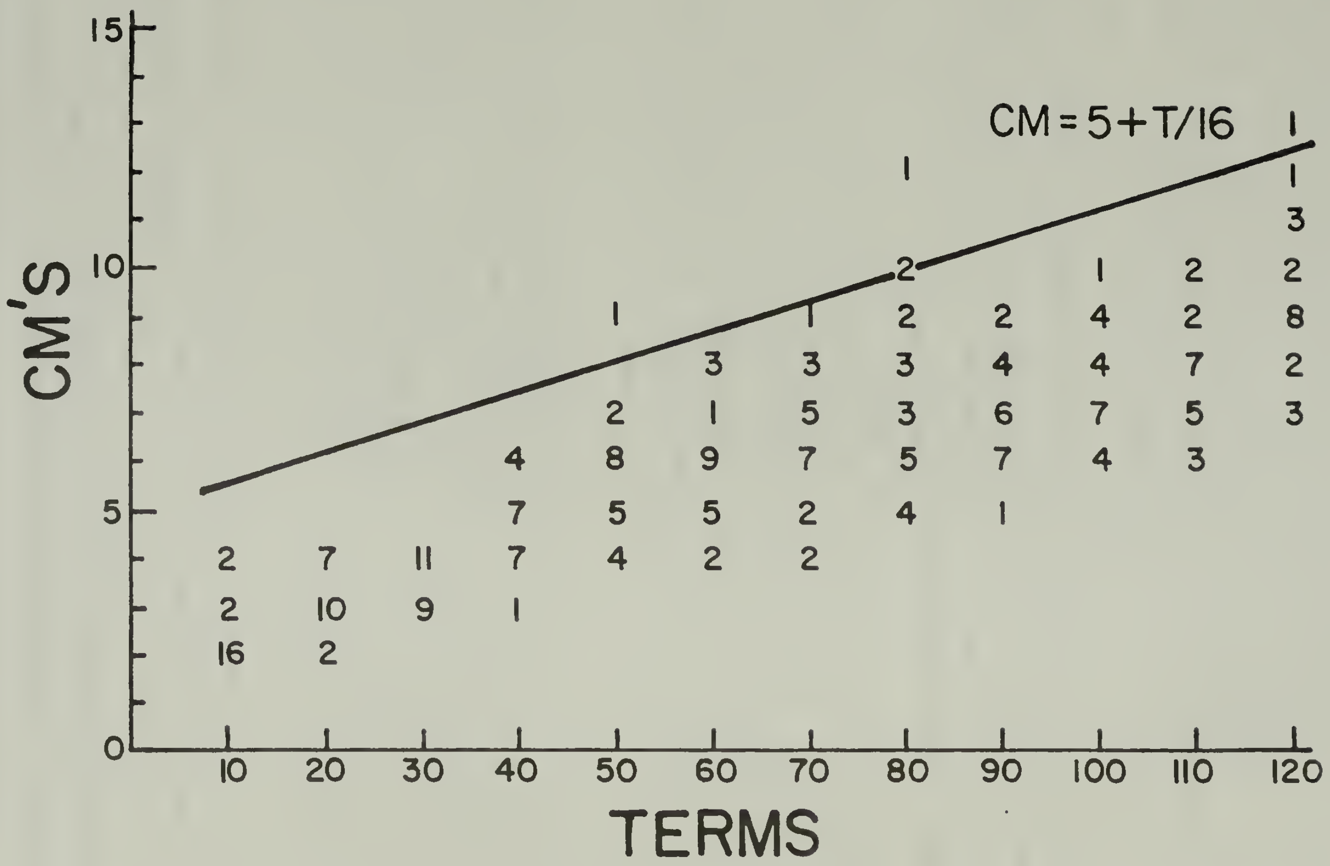


Figure 3.8 CM's vs. Table Size for Unlimited Interconnection (MAXINC)

3.2.5 Lower Bound on Number of Neighbor-Connected CM's

Figure 3.8 showed how many CM's were necessary to match tables of various sizes under the assumption that any CM could fork to any other CM. A practical implementation of the NFSA searcher requires limiting the interconnectivity, so it is necessary to determine whether this limitation results in an increase in the necessary number of CM's.

Section 3.2.2 outlined a method for computing minimal CM assignments. It involved building the pair table, computing the maximal compatibles, and choosing a cover which did not violate the interconnectivity constraints. The incompatibility cover and term tail removal algorithms developed in Section 3.2.3 can be applied to speed up building pair tables, with the note that if a term appears in more than one block, extra checks have to be made to insure that its states are assigned to the same CM in both. However, since most states are compatible, computing maximal compatibles takes much more time and space than computing maximal incompatibles. Finally, choosing a cover is still a problem.

Examination of some of the tables generated by runs of MAXINC showed that they could be assigned to the minimum number of CM's even if they were only neighbor-connected. The fact that without exception the maximal incompatibles included only startup states seems to indicate that the number of intersecting startup transitions in the table is the limiting factor on the number of CM's.

To test this theory without going to the trouble of implementing a program to perform assignments using covers of maximal compatibles, it was decided to run MAXINC to determine the minimal number of CM's necessary with unlimited interconnection, and then supply this number to the first-fit algorithm to be described in Section 3.3 to check whether the table could be assigned to the minimal number of matchers with neighbor interconnection being enforced. The program only failed to come up with an assignment for one of the 240 random tables (one with 110 terms). Thus, the results of running a maximal-compatible best-fit program would be essentially the same as those shown in Figure 3.8. This experiment vindicates the neighbor-connected CM organization.

3.3 First-Fit Assignment

When an NFSA matcher is configured for a particular system, techniques discussed in the previous section will be used to decide how many CM's each matcher will have. During production operation of the system, it is not necessary to compute minimal-CM assignments, it is merely necessary to find the first assignment which fits into the available number of CM's. Offsetting the apparently easier nature of this task is the fact that for large systems requiring search hardware, many users will simultaneously be searching the database, generating queries at a relatively high aggregate rate. This requires computing state assignments very quickly if the system is to maintain adequate response.

There are only three requirements for a valid first-fit assignment. First, no incompatibles can be assigned to the same CM. Second, states can only make transitions to other states in the same or neighboring CM's. Finally, no more states can be assigned to a CM than will fit in its memory.

The method that will be used to generate state assignments is quite similar to that for computing maximal incompatibles - first the state table is divided into blocks, and then blocks are processed sequentially. States in each block are assigned to CM's such that compatibility and interconnectivity constraints are satisfied. If a state is assigned to more than one block, the first assignment made during processing one block is honored later when others are processed. Unassigned states in blocks are assigned `around' ones that already have addresses.

3.3.1 Compatibility Testing

It is still necessary to determine pairwise compatibility as for best-fit assignment, and the same optimizations that were developed for best-fit apply to first-fit. All states in each maximal incompatible had to appear together in some block if all of them were to be found. For first-fit, it is only necessary that each incompatible pair appear together somewhere. In theory, this allows more freedom in generating cover blocks, and suggests the possibility of building smaller blocks than

for best-fit. Care must be taken not to pursue this strategy too far. CM's are assigned when a block is processed; it is possible for two incompatible states residing in more than one block each to be inadvertantly assigned to the same CM before the block in which they reside together is processed. The mistake would be caught, but unless logic was included in the assignment program to `back up' and re-process previously completed blocks to change the assignment, the only action possible would be to abort the assignment and signal the error. Therefore, Algorithm INCOVR from Section 3.2.3.1 is used as is. Experience shows it to provide good efficiency in incompatibility testing and result in a negligible number of aborted assignments.

3.3.2 Assigning States to CM's

Once the pair chart for a block is computed, all unassigned states are assigned to CM's. It was decided to try a rather simple assignment algorithm first, rather than to spend time implementing sophisticated heuristics that might not be needed. Therefore, a simple recursive backtrack algorithm was implemented in PDP-11 assembly language. An impressionistic high-level rendition of the actual program is shown in Figure 3.9

For the reader not ambitious enough to wade through the program, perhaps a brief prose description is in order. Essentially, ASCVR processes each block in the cover. It first calls a routine from MAXINC (FNDCMP) to build the pair table. The number of CM's available is computed as a function of the number of terms (or alternatively is computed by generating the maximal incompatibles). Then, ASBLK is called to assign the states in each term in the block to CM's. ASBLK just sets up calls to ASSIGN, a recursive procedure which assigns a state, its forks, and its successors to CM's. ASCVR and ASBLK vary the initial trial CM from term to term to attempt to balance the number of states equally among the CM's. ASSIGN uses the CM passed as an initial trial value; if the state cannot be assigned to that CM, all other possibilities are tried. For startup states, this includes all CM's; for forks, only neighbors are allowed as alternates. If the state to be assigned is

```
PROC ASCVR;                              ;ASSIGN ALL BLOCKS OF COVER
 CM<-0;                                  ;START ASSIGNING TO CM 0
 DO FOR I=1,NBLKS;                       ;DO FOR ALL BLOCKS
  (COMPUTE PAIR TABLE)
  BLOCK<-CVRBLK(I);
  NTRM<-CVRSIZ(I);
  ASBLK(BLOCK,NTRM,CM);                  ;ASSIGN BLOCK TO CM
  CM<-MOD(CM,NCM);                       ;BUMP CM # MOD NO. OF CM'S
 END;
END ASCVR;


PROC ASBLK(BLOCK,NTRMS,CM);
 DO FOR I=1,NTRMS;                       ;DO FOR ALL TERMS IN BLOCK
  TERM<-BLOCK(I);                        ;NEXT TERM IN BLOCK
  STATE<-TERM(1);                        ;START W/1ST STATE IN TERM
  ASSIGN(STATE,CM);                      ;START TRYING AT CM
  CM<-MOD(CM,NCM);                       ;NEXT TERM AT NEXT CM
 END;
END ASBLK;


PROC ASSIGN(STATE,CM);
 DCL DTBL(16) INIT(0,1,-1,2,-2,3,-3,...); ;ORDER TO TRY CM'S
 IF (STATE=NULL) THEN RETURN;
 IF (CM(STATE) .NE. NULL) THEN RETURN;  ;IF ALREADY DONE
 IF (PREV(STATE) = NULL) THEN N<-NCM;
  ELSE N<-3;                             ;START STATE CAN GO ANYWHERE
 DO FOR I=1,N;                           ;TRY ALL VALID CM'S
  TCM<-MOD(CM+DTBL(I),NCM);              ; ... CLOSEST FIRST!
  IF (NO INCOMPATIBLES IN TCM) THEN DO;
   CM(STATE)<-TCM;                       ;SET TRIAL CM
   ASSIGN(NEXT(STATE),TCM);              ;TRY ASSIGNING REST OF TERM
   IF (ASSIGN WAS SUCCESSFUL) THEN DO;
    ASSIGN(FORK(STATE),CM)               ;ASSGN FORKS TO ORIG. CM
    IF (ASSIGN WAS SUCCESSFUL) THEN RETURN;
     ELSE DO;
      (DEASSIGN NEXT(STATE));            ;UNDO THE DAMAGE
      CM(STATE)<-NULL;
     END;
    END;
   ELSE CM(STATE)<-NULL;                 ;IF 1ST ASSIGN DIDN'T WORK
  END;
 END;                                    ;TRY NEXT CM
 FRETURN;                                ;FAIL IF NO VALID ASSIGNMENT
END ASSIGN;
```

Figure 3.9 State Assignment Algorithm

compatible with all currently in the CM, ASSIGN is called recursively to attempt assignment of the successor and fork states. If both succeed, ASSIGN returns to its caller successfully also.

If an assignment is possible, it will almost always be found very quickly. If the table will not fit in the allotted number of CM's, all possible placements of the incompatibles will be tested, and it will take a long time to decide that no assignment is possible. To combat this problem, the following tests were added to the assignment program:

1. If the total number of incompatible states in any block is greater than the available number of CM's, the maximal incompatibles of that block are generated to see if it is possible to fit the table into the matcher. The assignment is aborted if not.
2. Certain alternative paths with no hope of correcting a problem are not tried. Many `alternatives' are just permutations among the CM's of states in the same maximal incompatible. Those which are detectable are not generated.
3. In case all else fails, a time limit is put on the assignment procedure. If the limit is exceeded, the assignment is aborted.

The processing necessary to recover from aborted assignments is beyond the scope of this chapter, but in general what would be done is to remove the problem term(s) from the table and perform the search in two passes.

3.3.3 Assigning State Addresses in CM's

When all states are assigned to CM's, their CM addresses are assigned. ASSIGN could possibly perform this function (see above), but it was decided to do it independently. If ASSIGN were delegated this function, the overhead when an alternate path had to be tried would be increased; the address assignment would have to be `undone' along with the CM assignment. It is necessary to traverse the terms table anyway to compute statistics at the end of the run, so doing address assignment during this pass is no extra trouble.

Transition states are assigned from the bottom of each CM's memory space, and forks are assigned down from the top. If a CM either has too many total states or too many forks assigned to it, an error is flagged. Admittedly assigning addresses during CM assignment would allow correcting these errors rather than aborting because of them. Attempting assignment to a full CM is identical to attempting assignment to a CM containing an incompatible state. The algorithm could back up similarly in each case. However, the number of CM's needed expands fast enough with increasing table size that there was plenty of room left in each CM even with the simple assignment algorithm used. Unless this fact changes, a more complicated address assignment algorithm is unnecessary.

3.3.4 AS2CMZ Program Overview

AS2CMZ is very similar to MAXINC, which was described earlier. The only difference is that instead of (or if desired, in addition to) finding the number of CM's by generating maximal incompatibles, the states are assigned addresses in CM's using the above methods.

AS2CMZ is essentially a test program, and the output available is similar to that of MAXINC. Figure 3.10 shows sample output from a run. Instead of printing the cover blocks and pair tables, the entire state table is printed out after addresses have been assigned. The statistics printed at the bottom are the same as for MAXINC. In this example the # CM's NEEDED is determined by computing maximal incompatibles, but in the tests used to generate the graph in Figure 3.12, the above formula for the number of CM's is used. The COMPATIBILITY CHECKS are tests done by ASSIGN, not tests done during pair table generation. The ASSIGNMENTS field shows the number of assignments attempted; this will be identical to the number of states unless backtracking was necessary.

In Figures 3.11 and 3.12 are shown the results of running AS2CMZ on the same tables used to generate the statistics in Figures 3.5–3.7. Recall that each data point represents the average time to assign five tables of the size shown. Considering that ASSIGN uses a potentially very slow recursive backtrack algorithm to assign states to CM's, one might wonder how efficient the program is in practice.

AISM BEST BENT BUNT
BUNTED IST SCHISM

ADDR	PCT	CHR	PRV	NXT	FRK	CCH	CM	CMA	F	QTY	INDX
071410:	14	A	000000	071430	000000	071430	00	001	14	01	0000
071430:	01	I	071410	071450	000000	071450	01	001	00	01	0004
071450:	01	S	071430	071470	000000	071470	01	002	00	01	0010
071470:	01	M	071450	000000	000000	072150	01	003	00	01	0014
071510:	14	B	000000	071530	071710	000000	01	377	50	01	0014
071530:	01	E	071510	071550	071630	071710	01	004	00	01	0004
071550:	01	S	071530	071570	000000	071710	01	405	00	01	7777
071570:	01	T	071550	000000	000000	000000	01	406	00	01	7777
071630:	01	E	071510	071650	000000	071530	02	001	00	01	0000
071650:	01	N	071630	071670	000000	071550	02	402	00	01	7777
071670:	01	T	071650	000000	000000	000000	02	403	00	01	7777
071710:	14	B	000000	071730	000000	071510	02	004	10	01	0010
071730:	01	U	071710	071750	000000	000000	02	405	00	01	7777
071750:	01	N	071730	071770	000000	000000	02	406	00	01	7777
071770:	01	T	071750	072110	000000	000000	02	407	02	01	7777
072110:	01	E	071770	072130	000000	000000	02	410	00	01	7777
072130:	01	D	072110	000000	000000	000000	02	411	00	01	7777
072150:	17	I	000000	072170	000000	072170	02	012	10	01	0000
072170:	01	S	072150	072210	000000	072310	02	013	00	01	0004
072210:	01	T	072170	000000	000000	072230	02	414	00	01	7777
072230:	14	S	000000	072250	000000	072250	01	407	10	01	7777
072250:	01	C	072230	072270	000000	072270	01	410	00	01	7777
072270:	01	H	072250	072310	000000	072310	01	411	00	01	7777
072310:	01	I	072270	072330	000000	072330	01	012	00	01	0010
072330:	01	S	072310	072350	000000	000000	01	013	00	01	0014
072350:	01	M	072330	000000	000000	000000	01	414	00	01	7777

TOTAL # OF TERMS = 00007, TOTAL # OF CHARACTERS = 00033
00004 UNIQUE 1ST CHARS, 00001 FORK STATES, AND 00026 STATES IN TABLE
00022 COMPATIBLES FOUND, 00030 TESTED;
00012 INCOMPATIBLES FOUND, 00021 TESTED.
00030 COMPATIBILITY CHECKS, 00026 ASSIGNMENTS, # CM'S NEEDED IS 00003.

TIMES: BLDFK-00012 BLDCVR-00166 ASBLKS-00607

Figure 3.10 Sample Output from AS2CMZ

Figure 3.11 shows one measure of this - the number of unsuccessful assignments as a percentage of the total number of attempted assignments. In this case, an unsuccessful assignment corresponds to the `if (no incompatibles in TCM)' test failing in the ASSIGN procedure in Figure 3.9. As the graph shows, for tables of up to 120 terms fewer than 20% of the assignment attempts are unsuccessful. Another possible measure of ASSIGN's efficiency is the number of backtracks executed - that is, the number of times a state is assigned to a CM and later de-assigned because a successor or fork could not be assigned. It is interesting that no backtracks were ever necessary when the number of CM's available was sufficient to hold the test table. Of course, the number of backtracks would be immense if the table did not fit in the CM's, but the tests introduced in Section 3.3.2 detect and stop excessive backtracking.

Figure 3.12 shows the average execution times for AS2CMZ on the test tables. Again, Curve 1 shows the execution time with incompatibility covers only, and Curve 2 shows the time needed with incompatibility covers and tail removal. Notice that Curve 2 indicates that execution time seems to grow linearly for the range of table sizes shown, and that tables of up to 120 terms can be assigned in nearly a second each. Tables with 120 terms contain nearly 1000 states, and the stated goal has been handling tables of this size in real time. Whether this performance is sufficient depends upon the query arrival rate and processing strategy, which will be covered in a later chapter. However, being able to do state assignment at the rate of about a millisecond per state on a PDP-11/40 with 1.75 microsecond cycle time is certainly encouraging.

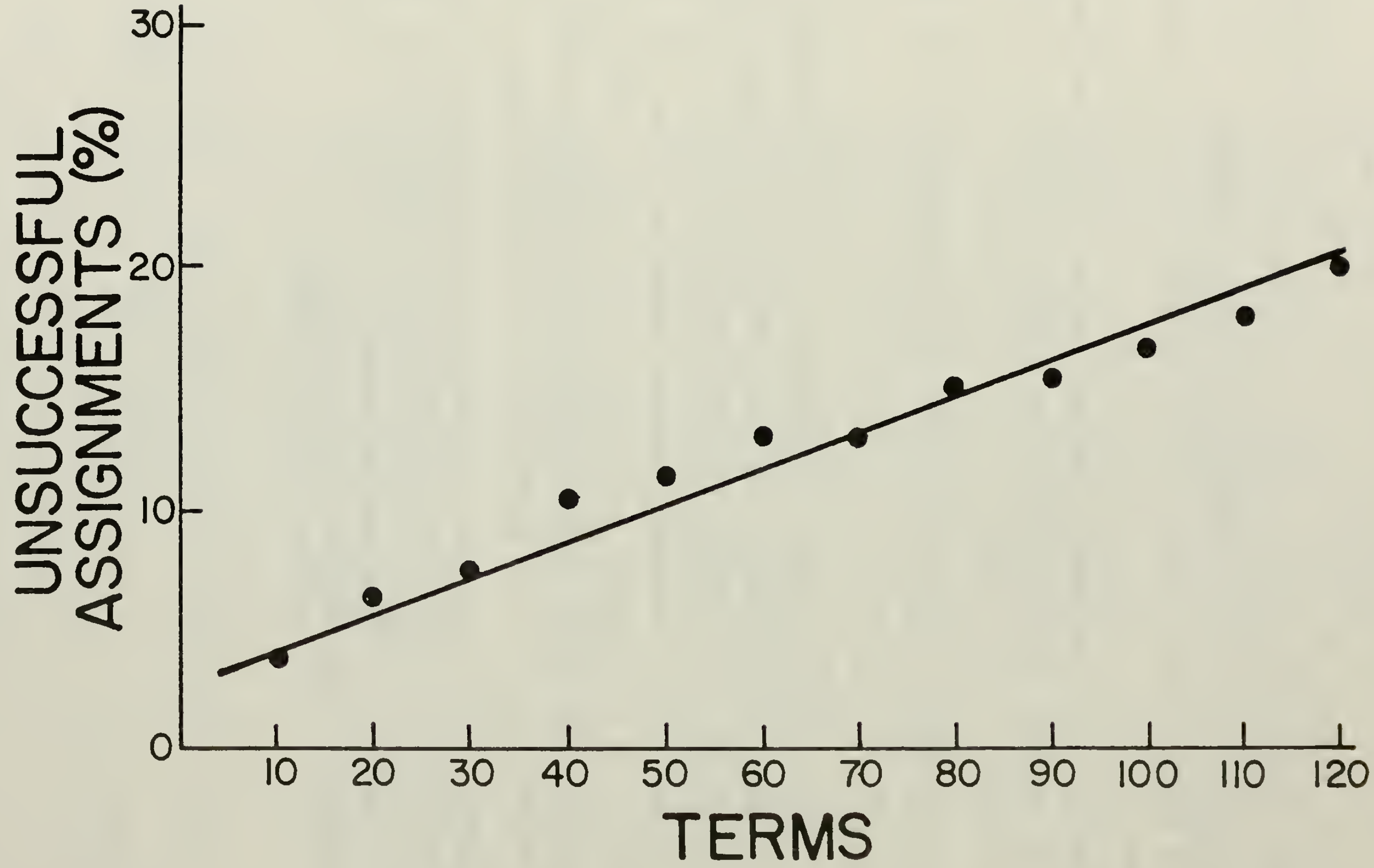


Figure 3.11 State Assignment Efficiency

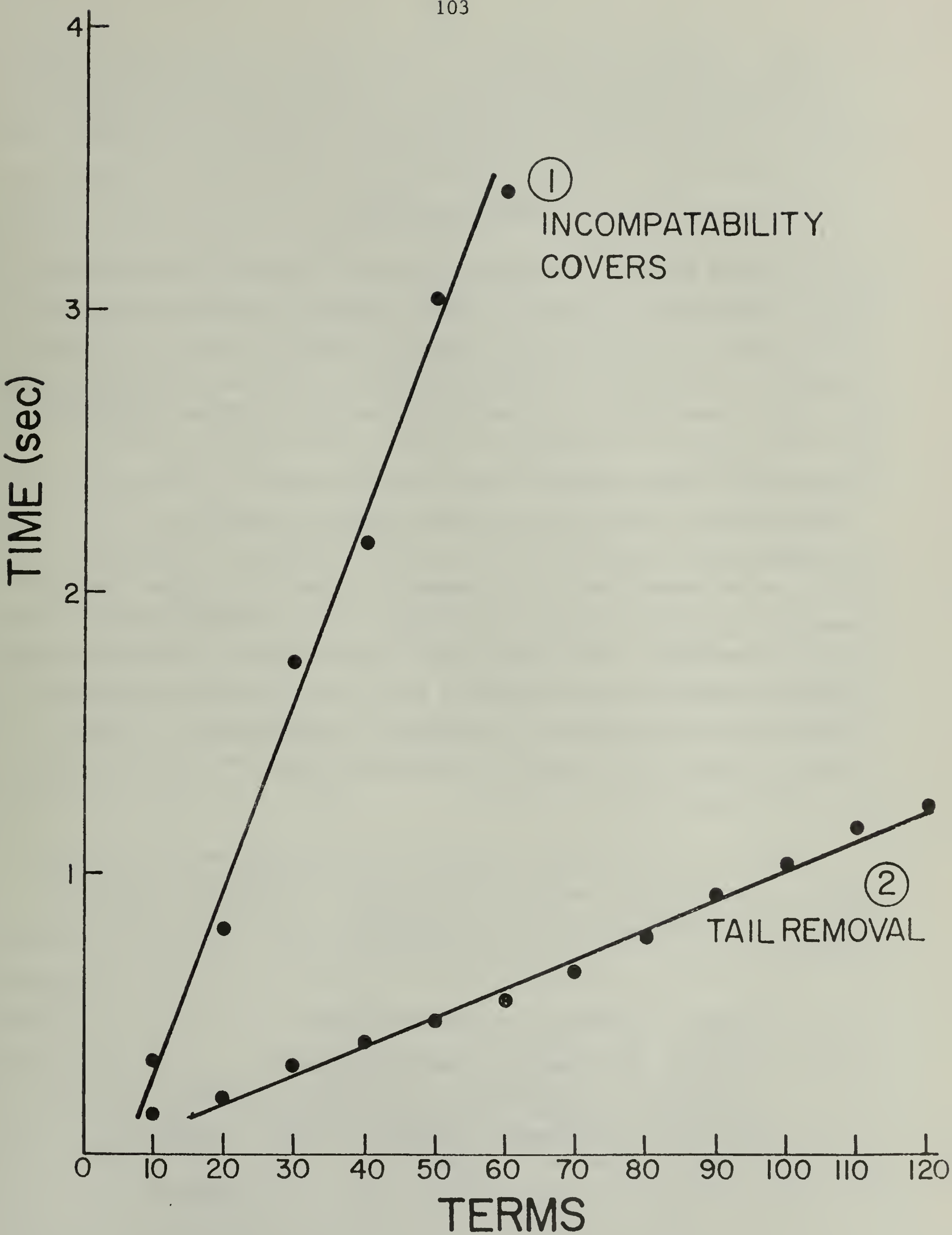


Figure 3.12 Number of Terms vs. Time for ASSIGN

Chapter 4

Query Resolution

The Query Resolver is responsible for accepting input from the Term Matcher informing it of the occurrence of items of interest in the text stream (terms, context boundary codes, etc.) and from these, detecting occurrences of the user's search expression. Figure 4.1 shows the syntax for search expressions in the query language used by EUREKA ([BurEm79]). The EUREKA language is reasonably powerful in the types of queries it allows while remaining fairly simple to learn and use. It will be used as a `lower bound' in the succeeding discussion; any successful implementation of the resolver must be able to handle it. Consider a sample search expression in the language: `(CHANDLER IN AUTHOR) AND (MARLOWE OR (PRIVATE AND DETECT? IN SENTENCE) OR (DEAD AND BODY IN PARAGRAPH) IN BODY)'. The term matcher detects instances of search terms (CHANDLER, MARLOWE, PRIVATE, DETECT?, etc.) and context boundaries (the AUTHOR section, the BODY text, paragraphs, and sentences). The query resolver performs the bookkeeping necessary to detect instances of the full search expression from these.

The structure of English-language terms is fairly well-defined and does not vary much from database to database. This made it possible to design special-purpose term searching hardware which would not have to be

```
<expr>::= <subexp> | <subexp><opr><expr>
<opr>::= AND | OR | & | +
<subexp>::= <trmexp> | <trmexp> <inclause>
<inclause>::= IN <ctxlst> | IN <ctxlst> <inclause>
<ctxlst>::= <context> | <context>,<ctxlst>
<context>::= SENTENCE | PARAGRAPH | AUTHOR | TITLE | ...
<trmexp>::= <term> | (<expr>)
```

Figure 4.1 Syntax of EUREKA Search Expressions

changed from system to system. Query resolution, on the other hand, is very much dependent upon the query language, and different languages can make fundamentally different requirements of the query resolver. It must be possible to program the query resolver to accept a wide variety of query languages, and this argues for implementing it with some sort of small general purpose processor (i.e. a microprocessor) rather than with special-purpose hardware. On the other hand, term hit rates can get quite high. If two terms were found in a typical 200-character sentence, the resolver would have 67 microseconds to handle each of the three hits. Unless careful attention is paid to the resolution algorithms, a general purpose processor may not be able to keep up.

Many specifics of query resolution are system-dependent, and will not be discussed in detail. However, the speed of the hardware term matcher and the desire to search each cylinder in one revolution pose problems for the query resolver which will not change from system to system. It is these system-invariant problems that will be discussed. First, a basic description of the query resolution process will be given. Next, considerations relating to the speed and generality of the resolver will be discussed. Finally, methods of handling documents spanning track boundaries and of processing queries too large to be searched for in one revolution will be introduced.

4.1 Query Resolution Algorithms

One query resolution algorithm which has been documented in detail is the one for the CIA's proposed SAFE full-text retrieval system, which is documented in detail in [OSI77b]. This system uses a hardware term matcher (a Bird FSA) for each disk, but searches only one track of the disk at a time rather than being able to search several (or all) tracks on a cylinder in parallel. Also, the search expression syntax is considerably less general than that of EUREKA. The language essentially allows only product-of-sums expressions, with proximity operations allowed only between <u>adjacent</u> terms in the outer product expression. However, there is enough similarity between SAFE and systems of the type being considered here to warrant a brief discussion of the OSI resolver.

The OSI Query Resolver (QR) receives input from the Term Matcher signalling the occurrence of search terms and document formatting codes (context delimiters). For each hit, the QR is passed a code which is an index into a table of addresses. Each address in this table (TTABLE) is a pointer to an entry in another table (XTABLE). The XTABLE entry contains one element for each query in which the term causing the hit appears. Each element points into a data structure (the QLB or query logic block) containing the internal representation of the search expression. When a hit is detected, the QLB entry corresponding to the term is marked as present. At the end of the document, the QLB is examined to see if the search expression was satisfied. If it was, the document number is saved to be later reported to the host.

The SAFE system query language is simpler than EUREKA's in that no context-bounded subexpressions (i.e. `<expr> IN SENTENCE') are allowed. The only in-context operator is proximity (i.e. `A AND WITHIN 3 WORDS B'), which is only allowed at the outermost level. The QR must differ from OSI's algorithm, since to handle complex context-bounded queries, resolution cannot be deferred until the end of a document.

Stellhorn ([Stell74a]) discusses query resolution for an experimental precursor to EUREKA. This system did term matching differently than ones that have been discussed in that one pass was made over the data for each search term. Text between two context delimiters (e.g. a sentence) was buffered, and searched for each term. When a hit occurred, the term's presence bit was set in the data structure representing the search expression, and the structure was scanned to see if the expression had been satisfied. If it had, a hit on the expression was reported.

Even ignoring the fact that Stellhorn's algorithm requires multiple passes over the data, it is not capable of resolving the EUREKA language. It can only identify and search one context at a time, and therefore cannot handle expressions like `(ENEMY AND AIRCRAFT IN SENTENCE) AND (SIGHT? AND OVERHEAD IN PARAGRAPH)'. On the other hand, it does allow context specifications to be attached to the entire expression rather than just allowing proximity specifications between adjacent terms.

Allowing context specifiers on subexpressions greatly increases the power of the language. For example, concepts can be expressed by constructs such as `DATA AND BASE IN SENTENCE', and used as terms in more complex expressions. However, a price is paid for this feature. Doing resolution for the full EUREKA search expression syntax requires considerably more logic than either of the above methods. The primary difference is that different subexpressions may be looked for in different and potentially overlapping contexts. Each time a context boundary mentioned in an `IN' clause of a query is crossed, the QR must check if the subexpression was found and then reset the term found flags for the subexpression. This is considerably more complex and time consuming than the OSI algorithm (which, except in the case of word proximity) only had to do any processing at document boundaries, and also more complex than Stellhorn's program, which reset <u>all</u> flags each time a boundary was crossed.

The EUREKA query resolver (written by the author) basically works as follows:

1. The search expression is represented by a tree. Each operator node contains context specifications and `found' bits for the right and left hand sides. These can be either terms or other operators.
2. Each time a term is found, it is looked up in a table. The table entry points to each node in the tree corresponding to an instance of the term. The proper `found' bits are set in the tree. Then the tree is traversed to see if the entire expression was satisfied as a result of the latest term hit. If so, the expression hit is reported.
3. When a context delimiter (fern) is encountered, the `found' bits are reset for subexpressions being searched for in contexts delimited by the fern.

Searching in EUREKA is done in software, and data is read into memory before being searched. Therefore, timing is not critical. If the EUREKA resolution algorithm were adopted for use with a hardware term matcher,

several problems might result. First, traversing the entire tree for each term found is too slow for large queries. Second, it is also necessary to go through the whole tree to reset found bits upon receipt of each fern. Both of these problems need to be corrected before the algorithm can be used to process output from the hardware term matcher in real time.

4.2 Query Resolution in Real Time

The speed at which query resolution can be done depends upon knowledge of expected term and delimiter hit rates. These rates in turn depend upon the number of queries being searched for, the complexity of these queries, and the frequency of occurrence of search terms in regions being searched. It was hoped that EUREKA could be used to measure these for a large sample of test queries, but the lack of a suitable database and a sizable user community made this impossible.

Little data is available regarding performance statistics of full text search systems. Roberts ([Rob77]) gives a statistical estimate of term hit rates. It is based upon the Zipf curve (rank order vs. frequency of occurrence in the database) and a curve of rank order in the database vs. frequency of occurrence in queries. Although he admitted that the latter curve was empirically derived, Roberts contended that the product of the two curves was essentially a constant, and therefore the expected frequency of occurrence of a search term while scanning the database was a constant, which he determined to be around 10^{-5}. For Roberts′ example system (64 key words per query and 20 queries being searched in a batch) the expected hit rate was one every 1215 microseconds. For a batch of 120 terms, which is closer to the size which has been used as an example in previous chapters, Roberts′ formula predicts one hit every 11.6 milliseconds, or approximately one term hit per track.

In a partially indexed system, of course, each track being searched would be known to contain at least one term occurrence, and usually more than one. Additionally, search terms are often looked for in conjunction with one another. Two semantically related terms such as `DATA′ and `BASE′ can be expected to occur in proximity much more often than is suggested by their frequency of occurrence. An analytical formula for hit

rate would therefore be a complex conditional probability problem. Knowledge of probability distributions of co-occurrences of terms in both queries and the database would be necessary. It is not clear that this information could be obtained except from analysis of a large corpus of queries.

If a large corpus of queries were available, it would be possible to make hit-rate measurements by simulating the search and instrumenting the simulator to gather the necessary statistics. However, it is expected that fundamental differences between the search system being proposed here and conventional systems will result in substantial differences in the way searches are performed. Conventional systems are characterized by large search costs and slow response. Therefore, users attempt to get high recall by such techniques as the use of a large synonym dictionary (thesaurus) and complex search expressions. A system using search hardware as being proposed here could have much lower cost per search and extremely fast response. This would substantially alter the dynamics of the search process, facilitating doing the logical query as a number of small steps. Thus, query characteristics can be expected to be quite different for hardware-augmented systems than for conventional ones. It is not clear that measuring hit rates using a corpus of conventional queries would yield valid results. The best that can be done before an experimental hardware augmented system is built and tried in practice is to make educated guesses about query characteristics and hit rates. It seems reasonable that the average search expression will be smaller and the term hit rate therefore lower. However, even increasing the estimate given by Roberts′ formula by a factor of ten predicts only around one hit per millisecond when searching for 120 terms. This figure is almost certainly much higher than will be observed in practice.

The hit rate for ferns will most likely be much higher. For example, in the Brown Corpus database ferns occur at an average of one every 200 characters. Most of these are end-of-sentence ferns, and intuition suggests that most sentences will not contain any search terms even in relevant documents. Thus, it can be expected that the most critical real-time function of the query resolver will be fern processing.

4.3 Query Resolution for the Hardware Term Matcher

Individual systems may have different search expression syntax and different requirements concerning reporting search expression hits. Rather than forcing all systems to conform to an arbitrarily defined interface to the text searcher, it was decided to allow the query resolver's operation to be tailored to each system within a fairly broad set of guidelines. This requires that it be relatively easy to modify details of the resolution algorithm, which in turn suggests that the query resolver be implemented in programmable logic (e.g. a microprocessor of some sort) rather than with fixed logic.

Since many of the details of the resolution process are system dependent, it is not worthwhile to present a detailed discussion of a complete query resolution system. On the other hand, it is necessary to demonstrate that the basic approach to resolution in the hardware searcher environment will work, which means demonstrating that the query resolver can do its job in real time. Fortunately, most of the time-critical parts of the resolver do not change from system to system. Term and fern hits have to be accepted from the matcher and recorded, the data structure representing the search expression must be scanned to detect query hits, and the `found' bits must be periodically reset as context boundaries are crossed. Timing estimates for these functions would most likely remain valid for any system.

Since some program fragments have to be implemented in order to obtain execution time estimates, some assumptions are necessary regarding the type of processor used in the implementation. The processor is constrained to being rather small, both because it will most likely be packaged in the disk drive cabinet and because it is desirable for the size and cost of the resolution logic to be of the same order of magnitude as that of the term matching logic. Since a PDP-11 was available for use by this project, sample program segments were implemented for it. Although the PDP-11 is a 16-bit processor, this is of no great advantage in many of the code segments, and in fact several microprocessors are available having close to the same performance. In a production system, a

single component microprocessor such as the Intel 8048 - possibly with the addition of some outboard hardware (RAM, I/O support) - would be used.

4.3.1 Accepting Hits from the Term Matcher

The term matcher signals the query resolver for every term hit. The matcher contains a limited amount of output buffering capability provided by the FIFO in the match controller (mainly to allow two successive hits, such as a term followed immediately by a fern, to be handled). It is still quite possible for a burst of hits to occur faster than they can be processed sequentially, so the resolver must be interruped by the occurrence of a hit, and must then buffer it in its own memory, to minimize the latency between hit detection and when the hit is read from the FIFO.

Figure 4.2 shows an interrupt routine to buffer term matcher hits. BUFFER is a circular hit buffer, with input pointer INPNT and output pointer OUTPNT. FIFO and STAT are device registers in the term matcher interface. When an interrupt arrives, the buffer pointer is loaded and the hit is read from the term matcher FIFO. It is saved in the buffer, and the circular wrap-around test is made. The program then checks for the buffer being full, and goes to an error routine (not shown) if it is. Finally, the term matcher status register is tested to see if another hit is available, and the program loops to buffer it if so. When no more hits are available, the pointer is saved and the program returns from the interrupt.

Assuming that the query resolver is at the highest interrupt priority, the maximum interrupt latency would be from the time STAT is tested, through the return from interrupt, back into the interrupt routine, until the FIFO is read. This takes 19 bus cycles or 33.25 microseconds (computed using the 1.75 microsecond cycle time memory on the PDP-11 used). Thereafter, a hit can be read from the FIFO every 20 bus cycles (35 microseconds). The total execution time of the interrupt routine if only one hit is read per interrupt is 59.5 microseconds.

```
.MAIN.   MACRO VD07.8/40 21-AUG-80 21:46 PAGE 1

                        ;
                        ; TERM MATCHER INTERRUPT HANDLER - SAVE HITS
                        ;    IN CIRCULAR BUFFER.
                        ;
000000 010046 TMINT:    MOV     R0,-(SP)        ;SAVE R0
000002 016700           MOV     INPNT,R0        ;INPUT POINTER INTO CIRC. BUF.
000006 013720 5$:       MOV     @#FIFO,(R0)+    ;GET DISK ADDRESS OF HIT
000012 013720           MOV     @#FIFO,(R0)+    ; ... AND HIT CODE
000016 020027           CMP     R0,#BUFEND      ;CHECK FOR WRAPAROUND
 00022 103402           BLO     10$
 00024 012700           MOV     #BUFFER,R0      ;RESET POINTER IF SO
 00030 020067 10$:      CMP     R0,OUTPNT       ;CHECK FOR BUFFER FULL
 00034 001433           BEQ     ERROR
 00036 005737           TST     @#STAT          ;CHECK IF MORE HITS PENDING
 00042 100761           BMI     5$              ;GET THEM IF SO
 00044 010067           MOV     R0,INPNT        ;RESTORE INPUT POINTER
 00050 012600           MOV     (SP)+,R0        ; ... & R0
 00052 000002           RTI
```

Figure 4.2 Term Matcher Hit Interrupt Routine

The nominal speed of a 3330-type disk is 1.25×10^6 bytes per second. The MSI prototype term matcher from Chapter 2 could buffer eight hits in its FIFO. If the FIFO started empty, it would take nine hits within 61.25 microseconds (76 characters) to saturate the interrupt routine. Cases which could cause this include eight consecutive sentences containing no search terms, averaging under eight characters (plus fern) each, or one sentence 76 characters long containing seven search term hits (one hit every 11 characters). Neither of these cases are very likely.

The worst-case sustained hit rate that can be handled is governed by the length of the loop (35 microseconds). Assuming the FIFO were full, one hit could be accepted every 44 characters. This rate is substantially greater than the expected rate of occurrence of either ferns or search terms.

This discussion demonstrates that the query resolver can accept hits from the matcher at a reasonably high burst rate. It is now necessary to ascertain whether the resolution itself can be done faster than the <u>average</u> hit rate.

4.3.2 Term Hit Decoding

Each hit report from the term matcher contains the disk address at which the hit was detected and a unique code (the CM state address) identifying the hit. The disk address is not used during resolution, it is used only to specify the location of expression hits in reports to the host. The code is used to address a hash table which yields a pointer to a term table entry. A term table entry contains a pointer to the presence bit for each occurrence of the term in search expression trees.

A term matcher with sixteen 128-state CM's has a total of 2048 possible states, any of which can be a final state. Allowing 2048 hash slots would require too much memory, so fewer must be used. Assume that each CM has 128 states, and the state address in the CM is used as the hash code. In this case, up to sixteen hit codes (state addresses) can hash into the same slot. If all states are equally likely to be terminal states, a binomial distribution can be used to calculate the expected

number of hit codes hashing into the same slot. Since the average term is seven characters long, each state has a 1/7 probability of being a terminal state. Assuming (in the worst case) that all states in all CM's were full, the expected number of hit codes hashing into each slot is two.

Figure 4.3 shows a program segment for taking a hit from the circular buffer filled by the hit interrupt handler and looking it up in the hash table. The expected value of the time to look up an entry is 42 microseconds, assuming it is equally probable that the hit code is any of those hashing to the same slot. It will take a couple of more instructions to determine whether the hit is a term or fern, and go to the proper routine.

4.3.3 Context Boundary (Fern) Processing

In the EUREKA query resolver, each time a context boundary is crossed the search expression tree is scanned for nodes whose context is restricted to contexts corresponding to the boundary just crossed. All \`found' bits in such nodes are reset so the search can begin anew the next time such a context is entered.

Since ferns occur on the average every 200 characters (160 microseconds) and since a substantial fraction of this time is already consumed handling the hit interrupt and looking it up in the hash table, visiting all nodes in each search expression to reset found bits is much too slow. For the same reason, it is not practical to look for search expression hits at fern boundaries (as opposed to during term hits). There are many more ferns hits than term hits, and it would be necessary to examine the tree after each fern hit even in the usual case of no term hits occurring between successive ferns. Since the Section 4.2 indicated that term hits occur so much less frequently than ferns, fern processing should be optimized, not term processing.

Thus, the primary thing that has to be done for each fern is resetting found bits. This can be substantially speeded up if the bits are not stored in the search tree itself. Instead, all found bits for nodes (subexpressions) being searched for in a particular context are

```
.MAIN.   MACRO VD07.8/40 21-AUG-80 21:46 PAGE 2

                       ;
                       ; ROUTINE TO FETCH HIT FROM CIRCULAR BUFFER AND
                       ;  LOOK IT UP IN THE HASH TABLE
                       ;
 000054 020067 LOOKUP: CMP     R0,INPNT        ;OUTPUT POINTER IS IN R0
 000060 001775         BEQ     LOOKUP          ;WAIT IF NO HITS PENDING
 000062 012005         MOV     (R0)+,R5        ;DISK ADDRESS FROM BUFFER
 000064 012001         MOV     (R0)+,R1        ;HIT CODE
 000066 010067         MOV     R0,OUTPNT       ;UPDATE TO SHOW HIT ACCEPTED
  00072 006301         ASL     R1              ;GET CM IN HI BYTE, ...
  00074 010102         MOV     R1,R2           ; ... HASH INDEX IN LOW
  00076 000302         SWAB    R2              ;CM INTO LOW BYTE OF R2
  00100 042701         BIC     #^C376,R1       ;ISOLATE HASH INDEX (STATE)
  00104 016103         MOV     HASHTB(R1),R3   ;GET ADDR OF 1ST ENTRY
  00110 120263 10$:    CMPB    R2,CM(R3)       ;SEE IF RIGHT ENTRY
  00114 001403         BEQ     FOUND           ;TERM FOR THIS HIT CODE
  00116 011303         MOV     (R3),R3         ;IF NOT, TRY NEXT
  00120 001373         BNE     10$             ;CHECK ALL TERMS IN THIS SLOT
  00122 000400         BR      NOTFND          ;SERIOUS ERROR- HIT NOT IN TBL!
```

Figure 4.3 Hash Table Lookup Routine

stored together. Each search tree node contains the address of a bit in one of these areas. Each time a fern is processed, all bits in the fern's area are cleared at once. Some ferns delimit more than one context (i.e. paragraph ferns also delimit sentences), and for such ferns more than one area is cleared. Most ferns delimit only sentences, so for the majority of hits only one area will have to be cleared.

The found bit areas can either be stored in main memory or in small, independent memory. If main memory is used, machine instructions must be used to clear the area. Each instruction clears a full word of bits (16 on the PDP-11) but even so, for large search expressions several microseconds may be necessary to clear a sufficient number of words. If outboard memory is used, the device could be chosen (or designed) to allow all bits to be cleared simultaneously (e.g. the `clear' function being activated as an I/O operation). Alternatively, logic could be used to access the memory and clear the bits (via DMA) simultaneously with CPU operation.

Assuming the found bits are stored in main memory, consider how much time would be necessary to reset them. For the benchmark batch of 120 terms, there could be a maximum of 240 found bits (one for each term and one for each search tree node). These occupy 15 PDP-11 words, which take a minimum of 52.5 microseconds to clear.

The total time to process a fern on the example PDP-11 is therefore the interrupt handling, hash lookup, and bit resetting time, which is 59.5+42+52.5 = 154 microseconds. This is under the average 160 microseconds between ferns, although not by much. However, this time represents the worst case, and it is in the right order of magnitude even using very slow 1.75 microsecond memory. Using faster memory or other techniques such as bit-reset hardware or DMA transfer of hits from the term matcher would bring fern processing time down significantly.

4.3.4 Term Hit Processing

What CPU time is left after processing ferns is available for processing term hits. The fern processing time depends upon whether

special hardware is used (i.e. the bit-reset feature), and it is not known exactly what the term hit rate will be (even though the one millisecond estimated in Section 4.2 should be a reasonable maximum). The necessary hit processing speed is a function of both of these undefined parameters, so there is no point in calculating exact routine timings. However, techniques will be discussed to help process hits quickly.

Each time the EUREKA query resolver receives a term hit, it marks the found bit and then does a top-down traversal of the search tree to determine whether to generate a query hit. Query resolution consumes only a fraction of the EUREKA text searcher′s time (term matching takes the largest percentage) so it is not important for the query resolver to be very fast, and not too much attention was paid to optimizing the routine. It is not surprising, therefore, that much can be done to speed the algorithm up.

Doing the resolution top-down requires that most of the tree be traversed for each hit. However, the number of nodes visited can be reduced substantially by resolving bottom up. Each time a hit occurs, the found bit is set and (for each node in which the term appears) the operator node is checked to see if it is satisfied. If it is, its found bit is also set and the next node up the tree is checked. Finally, when the root node is marked found, a search expression hit is reported. For a tree with N operator nodes, only $O(\log_2 N)$ nodes are visited each hit.

Complicated synonym expressions (for example those generated by a thesaurus or phrase dictionary) will appear as large subexpressions connected by OR operators. Figure 4.4 shows such a subexpression. Naturally, finding any of the terms results in the highest OR node in the subtree being satisfied, but a simpleminded bottom-up algorithm will traverse all nodes between the leaf and the subtree root. However, by setting the up-pointer of each leaf in Figure 4.4 to point to the subtree root results in the intermediate nodes being skipped and the root being marked found immediately. In effect, the subtree root becomes a node of of an n-ary subtree. Although the leaves in this case are illustrated as terms, there is no reason why they cannot also be subtrees.

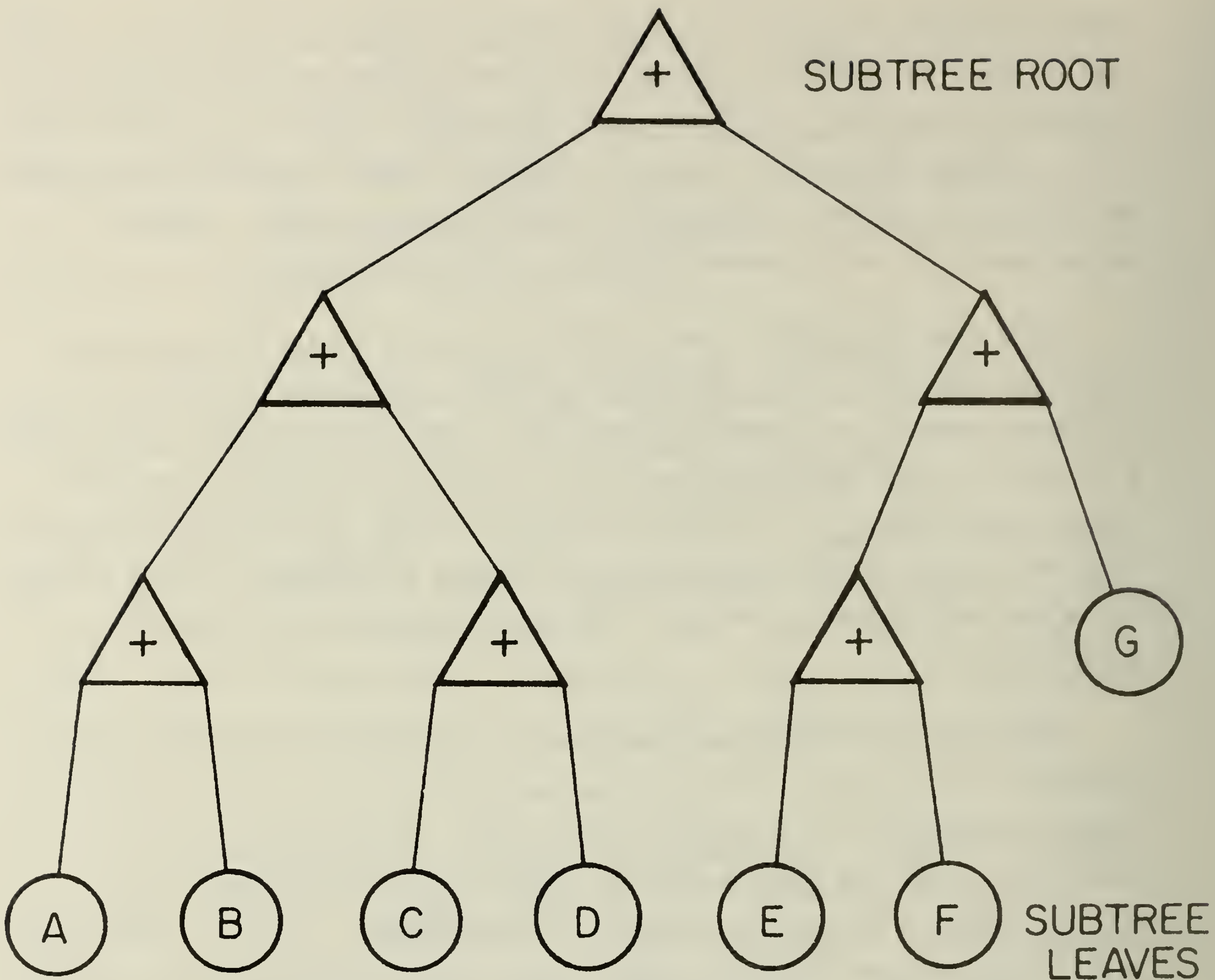


Figure 4.4 OR-Subexpression Tree

A similar technique can be applied to large blocks of AND's, even though these will probably not occur very often (AND's are normally used to specify phrases, such as `ROCK AND ROLL', and phrases normally consist of only a few terms). Figure 4.5 shows an example. In this case, the subtree root contains a counter and a field containing the number of leaves. Each leaf points up to the subtree root. Each time a leaf is found, the counter in the root is incremented. The subexpression is satisfied when the counter equals the number of leaves. The counter must be cleared each time the found bits are cleared, and doing this might substantially slow down fern processing. This plus the infrequency of large groups of AND's mitigate against including this optimization.

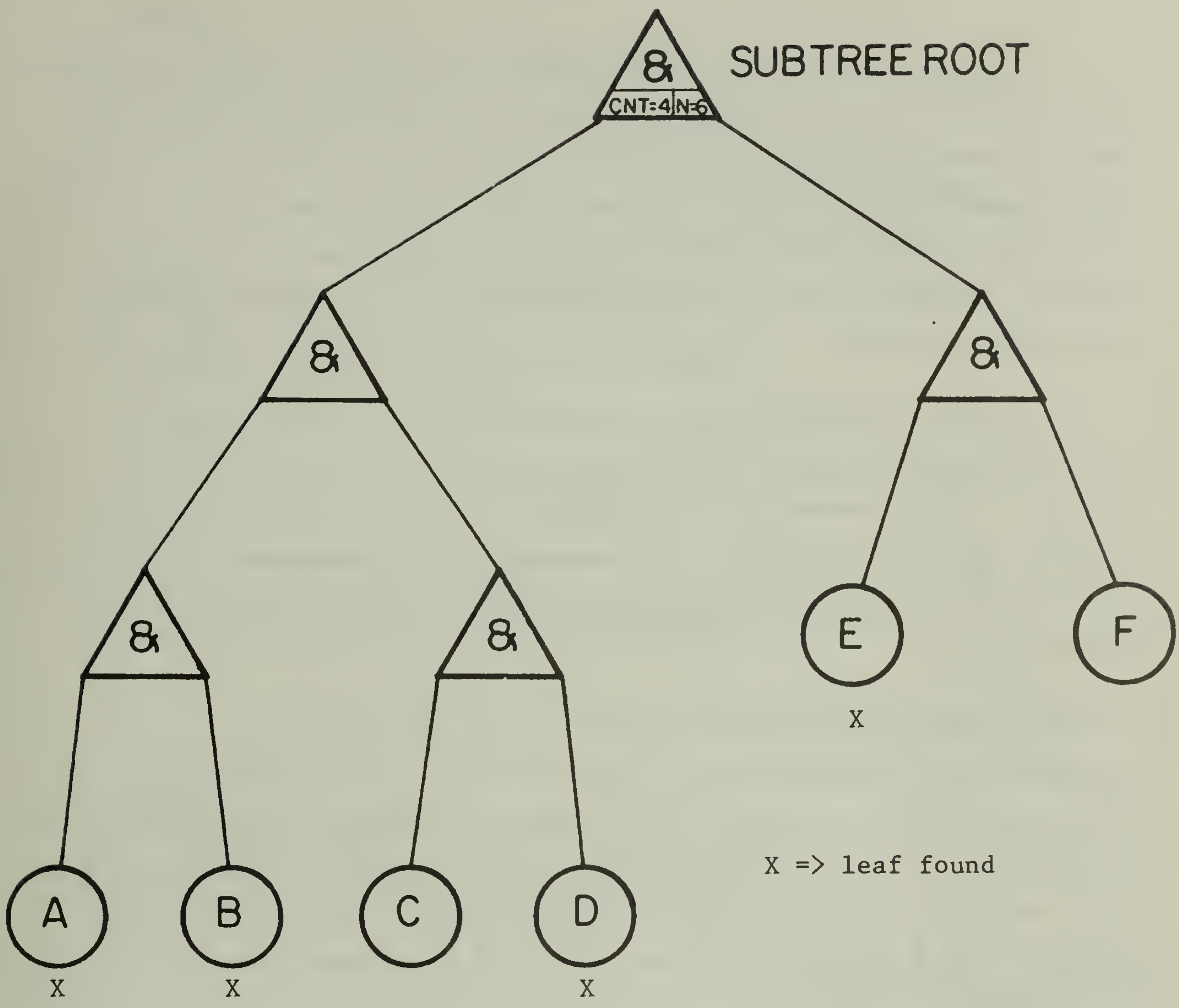


Figure 4.5 AND-Subexpression Tree

4.4 Contexts Spanning Track Boundaries

If documents were wholly contained in one track, searching could be done by examining tracks independently. Organizing the data like this is not practical, however, because documents are large relative to the track size. Not only would space be wasted at the end of each track due to fragmentation, but many documents will not even fit on a track. It is therefore necessary that documents, and even the larger contexts within documents (e.g. sections), be allowed to reside on more than one track. This introduces the problem of searching for queries in contexts mapping into more than one track (i.e. `A AND B IN DOCUMENT').

The basic strategy for handling contexts spanning track boundaries is to search the track containing the start of the context, and save the partial results (found bits) existing at the end of the context fragment. These results are then used when searching the continuation of the context on the next track. For contexts residing on more than two tracks, the partial results are passed along until the track containing the end of the context is searched.

The manner in which data is organized on disk affects how this strategy is carried out. Two possible ways that spanning contexts can be arranged include continuing the context on the next track of the same cylinder (vertical spanning), or continuing it on the same track of the next cylinder (horizontal spanning). Each choice has advantages and drawbacks.

An example of vertical spanning is shown in Figure 4.6. One paragraph starts on track 0 and ends on track 2, and another starts on track 2 and ends on track 3. Both contain search expression hits. The right arrow at the end of tracks 0, 1, and 2 represent a fern indicating that the context is continued on the next track. Similarly, the left arrows at the beginning of tracks 1, 2, and 3 indicate contexts continued from previous tracks.

During searching, the query resolver saves the found bits for contexts only partially contained in the track. These are combined to resolve the entire expression. For example, QR_0 sends a message to QR_1 saying that it has found `A'. The paragraph continues through track 2 and onto track 3, so $QR_0$ combines its own partial results with those from $QR_0$, showing that both `A' and `B' have been found, and passes them to QR_2. The paragraph ends in track 2, so QR_2 saves the partial results when the end of the continued paragraph is encountered. When these are combined with the results from QR_1, a hit on the entire expression is detected. Additionally, the partial results for the paragraph continued on track 3 are sent to QR_3, telling it that `C' has been found. The partial results saved by $QR_3$ show that `A' and `B' were found in the part of the paragraph on track 3, so combining them with the results passed from QR_2 produces

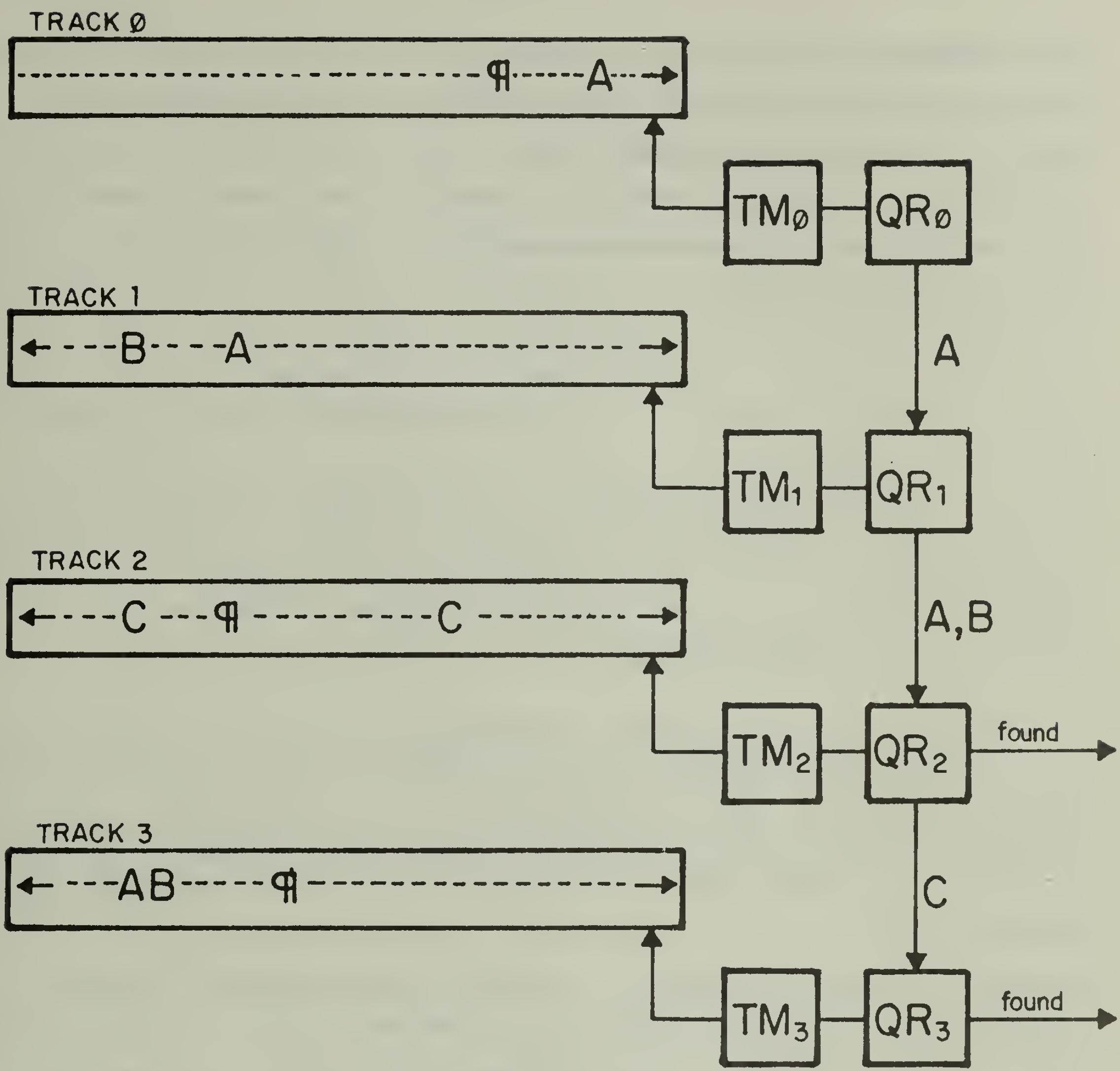


Figure 4.6 Resolution for Contexts Spanning Track Boundaries

another search expression hit. A QR only needs to save one set of partial results for each continued context (e.g. two sets must be saved if both a paragraph and a section span the track). Thus, the amount of memory required to save partial results is quite small.

As far as the query resolver is concerned, the context can be broken anywhere and each track can be completely filled, with no space wasted. However, indiscriminately breaking contexts causes problems for the term matcher if vertical spanning is used. Obviously, terms cannot be broken in two, or the matcher would never find them. Breaking the track at word

boundaries results in continuous word phrases being missed. If the query language imposes the restriction that CWP's must reside in the same sentence (or whatever the lowest level context is called in a particular system), and if sentences are not allowed to span tracks, term matching will work properly. Sentences are typically short enough that a minimum of disk space is wasted.

If horizontal spanning is used, the continuation will be on the same track of the next cylinder. It will be processed by the same searcher, and the continuation can be treated as an extension of the same data stream. Up to one full surface of the disk (12800 Kbytes on a 3330) can be treated as a continuous data stream. In fact, if the CM's in the term matcher are left in the same state during seeks, individual terms can be broken across track boundaries.

Horizontal spanning has the advantage of allowing the data to be split at any point. The only space wasted is at the end of a surface, as documents cannot be split across surfaces. Another advantage is that no facility for passing partial results from QR to QR is necessary. One disadvantage is that when system load is light enough that not every cylinder contains the start of a search region, horizontal spanning can require searching more cylinders and therefore can slow response time. Another problem occurs if more surfaces on a cylinder require searching than there are searchers available. In the scheme described above, a searcher remains connected to one disk head as long as the context it is searching continues. It is therefore necessary either to back up to a previous cylinder to search the other surfaces, or to delay searching them until the next scan.

Vertical spanning overcomes these problems, since it is possible to search all contexts on a cylinder without seeking. However, to avoid staying on a cylinder for more than one revolution, the drive must have one searcher for each track occupied by a long context, in addition to a facility for passing partial results from searcher to searcher.

Whether horizontal or vertical spanning is indicated for use on a particular system depends upon several factors. If most cylinders

required searching, and if several searchers per drive are available (such as in a non-indexed system), horizontal spanning would allow more efficient utilization of disk space with little or no penalty in response time. However, if the system load were such that queries were normally processed sequentially rather than being batched, vertical spanning would increase the load which could be accomodated before saturation occurred, assuming that a sufficient number of searchers per drive were available.

4.5 Processing Large Search Expressions

It may sometimes be impossible to process a track in one disk revolution, usually because the queries being searched for contain too many terms to be assigned to the available CM's. Three strategies exist for processing such query batches in multiple revolutions.

The simplest is to divide the batch of queries into two or more smaller batches and load and search each batch separately in several revolutions. This is easy to do and requires no extra capability in the text searcher. However, this strategy fails if one individual query is too large to fit.

A strategy for large expressions which will not fit into the searcher is for the host to break them up into two or more subexpressions and search each subexpression during one revolution. A hit on a subexpression includes the hit's location (document number, section, paragraph, and sentence number), which can be used by the host to detect hits on the entire search expression. This method requires the host to store the hits for each revolution, and to merge them after the last revolution to resolve the query. If many searchers are active simultaneously, this may put a heavy load on the host.

It is also possible to handle multi-revolution queries entirely in the query resolver in a manner completely analogous to processing contexts spanning track boundaries. The entire search tree is loaded into the query resolver, but only a portion of the terms are loaded into the term matcher for each revolution. At the boundary of each context in which a subexpression is being searched for, the found bits are saved. During

successive passes, the partial results from previous passes are combined with hits from the current revolution. Search expression hits are detected in the usual manner, and by the end of the last pass all hits will have been reported.

If ferns occur on the average every 200 characters, there will be 82 ferns per track. There would presumably be more terms than the term matcher was designed to handle, so quite a large amount of memory would be necessary to store partial results (20K bits for 120 terms). Also, saving and restoring partial results at fern boundaries could push the query resolver CPU into saturation. Building the query resolver with enough speed and memory to handle multi-revolution queries would result in its being much larger and more expensive than the term matcher.

Presumably a system would be configured to handle queries of up to a given size. Larger queries would be relatively rare, and including expensive hardware to facilitate handling them does not seem reasonable. For this reason, breaking large queries into subexpressions and doing final resolution in the host seems to be the best strategy at the present time.

4.6 Summary

Considerable effort was made in the previous chapters to develop term matching hardware which could be built cheaply in quantity. The aim was to make implementation of a search-intensive system economically feasible. Each term matcher must be connected to a query resolver. The two should have the same order of magnitude cost to avoid defeating the purpose of developing a cheap matcher. The most attractive implementation fitting both capability and cost criteria is one using a single-chip microprocessor.

The hit rate from the term matcher is considerably lower than the disk data rate, but is still high enough that careful attention must be paid to efficient implementation of the query resolution algorithms. The parts of the query resolution algorithm having the highest frequency of execution were identified, and techniques were discussed for optimizing

them. Sample routines were written for some of the critical sections, and timings were done to show that it is feasible to do query resolution on a relatively slow programmable processor.

Finally, extensions of the resolution algorithms were introduced to allow searching contexts crossing track boundaries, and to allow searching queries too large to be processed in one revolution.

Although system-dependent portions of the query resolver were not covered in detail, sufficient material regarding the system independent portions was presented to indicate how the most important operations are done, and to demonstrate that they can be done in real time.

Chapter 5

System Configuration and Performance

The component parts of the hardware text searcher have been discussed in previous chapters. It is now necessary to discuss how they will fit together into a system, and how that system can be expected to perform under load. Certain aspects of this have been touched upon previously. For example, it was shown how the expected number of terms being searched for affected the size of the term matcher. Many other parameters which vary from system to system also influence configuration and performance. Factors such as the query interarrival time, the indexing technique, and the disk speed all have implications on the internal design of the searcher. Query complexity and system load affect both the number of searchers needed per disk drive, and the time needed to load these searchers with state tables and search trees.

This chapter will try to quantify some of these system dependent variables, and in the process will try to provide some insight into how a system using hardware searchers would function. Term matcher characteristics (the required number of states per CM and CM's per matcher), searcher characteristics (number of searchers per disk drive, time needed to load the searchers), and behavior under load (scheduling techniques, response time, etc.) will be discussed. The aim is to show how these variables change for different types of systems. It will be shown that the hardware searcher which has been developed in preceding chapters can be used in a wide variety of environments, and that excellent response time can be maintained at a reasonable hardware cost.

5.1 System Behavior Under Load

The basic operation of the system was mentioned in Chapter 1. Queries are input by users to a host machine, index processing is done (if an index is used) to narrow the search to a number of candidate regions of the database, and these regions are then searched to see if they actually

contain instances of the search expression. Logically, the term `region' refers to a context of a document suspected to contain an instance of the search expression, but for purposes of this discussion, each region can be thought of as a disk track containing such a context.

The actual searching is done independently in each drive. Each drive can be configured with up to one searcher per read head, enabling any desired number of tracks on a cylinder to be searched simultaneously. Before a region is searched, the search tree must be loaded into the query resolver and the state table memories in the term matcher must be loaded. If multiple regions are to be searched for on a cylinder, all searchers must be loaded prior to beginning the search.

Once the search is underway, hit reports are buffered by the QR's and subsequently sent back to the host. The host collects hit reports for all queries coming back from each drive, and saves these for later presentation to the user. At the end of the revolution, a seek is initiated in each drive to move it to the next cylinder to be searched, and the searchers are reloaded as necessary. The search process is then repeated on the new cylinder.

The number of searchers, and to some extent the amount of hardware in each searcher, is influenced by the expected load on the system. Each system should be properly configured to be able to efficiently handle the expected maximum load using the minimum necessary amount of hardware. Therefore, an understanding of how loading effects such variables as search time, searcher capacity, and the necessary number of searchers is useful.

This investigation will provide the maximum amount of insight into the behavior of the system if numerical results are presented. To enable this to be done, it is necessary to make some assumptions about the implementation of the system. Unless stated otherwise, these assumptions are as follows:

1. The text is stored on 3330-type disks. For this type of disk, the typical minimum seek time is 7 milliseconds, the revolution

time is 16.67 milliseconds, there are 19 tracks per cylinder, and 800 cylinders on a disk.

2. The data is organized so tracks on adjacent cylinders begin 180 degrees out of phase from each other. This is possible if soft sectoring is used. A one cylinder seek can be done in just under 1/2 revolution, so this organization makes it possible to search a cylinder, seek to the next cylinder, and begin searching it almost immediately. This allows searching the entire disk in the minimum amount of time.
3. It is assumed to be possible to search as many tracks as necessary on a cylinder simultaneously. This means that a sufficient number of read head amplifiers, demodulators, and searchers are available to search the number of tracks on a cylinder being accessed by all queries being processed.
4. All drives in the system are searched simultaneously in parallel. Thus, it is only necessary to consider what happens in any one drive.
5. The probability that a given query will access a given track on a disk is uniformly some number p_a (which implies that all queries access a uniform fraction p_a of the tracks). This number reflects the selectivity of the indexing system.
6. The query interarrival time t_q is uniform.

5.1.1 Search Time

Each query accesses tracks with uniform probability over the entire surface of the disk, so it is assumed that a one-directional SCAN scheduling policy is used. During each scan, cylinders accessed by queries are scanned sequentially from the outside of the disk to the inside. When the innermost cylinder is reached (or when no queries being searched access inner cylinders), the head rack is moved back to the outside of the disk and another scan is started. This provides good response (at the expense of one long seek at the end of each scan) and eliminates the standard SCAN policy's bias against the middle cylinders.

At any cylinder, if the adjacent interior cylinder is to be searched, the seek will take under 1/2 revolution. Since the start of data on the tracks of this cylinder is displaced by 1/2 revolution, the seek and search can be done in 1.5 revolutions. If a two-cylinder seek is necessary, the data will be displaced by a full revolution, and two revolutions (one for the seek and one for the search) will be needed. A 3-cylinder seek will take over 1/2 revolution, so 2.5 revolutions will be necessary for the seek and search. For all but very long seeks of over one cylinder, one revolution will be long enough to complete the seek, so the expected time to seek and search the next cylinder is (2+2.5)/2=2.25 revolutions. If p_c=Pr[must search next cylinder], and R is the revolution time of the disk, the expected time to search the next cylinder is:

$$t_c=p_c(1.5R) + (1-p_c)(2.25R) = R(2.25 - .75p_c)$$

If the number of cylinders is C, each scan searches an average of p_cC cylinders. In this case, the average scan time is:

$$t_s=p_cCR(2.25-.75p_c)$$

The probability that a cylinder is accessed is the probability that any track on it is accessed, which can also be stated:

$$\begin{aligned} p_c &= 1 - \text{Pr[no tracks to search on cylinder]} \\ &= 1 - (1-p_t)^T \end{aligned}$$

where p_t is the probability that a track requires searching and T is the number of tracks per cylinder. The probability that a track does <u>not</u> require searching, $(1-p_t)$, is the probability that no query being searched accesses the track. The number of queries in the system is n, and p_a is the probability that a given query accesses the track, so:

$$(1-p_t) = (1-p_a)^n$$

and

$$p_c = 1 - (1-p_a)^{nT}$$

so the scan time t_s can be expressed as

$$t_s = CR(1-(1-p_a)^{nT})[2.25-.75(1-(1-p_a)^{nT})]$$

The time necessary to search all cylinders (i.e. for $p_a=1$) is $t_{max}=1.5RC$, which is 20 seconds for a 3330 disk drive (R=.01667 sec., C=800 cylinders).

5.1.2 Number of Queries in the System - Constant Interarrival Time

Each query remains in the system for one scan, so t_s is the time necessary to service all queries in the system. To solve for n, the number of queries in the system, consider that during any scan, $n=t_s/t_q$ queries will arrive. Thus, at equilibrium, $nt_q=t_s$, which allows solving for n. Figure 5.1 shows a graphical solution. Notice that t_s reaches a maximum of t_{max}, which is the time necessary to search all cylinders. Curves 1, 2, and 3 show nt_q for varying interarrival times t_q. For $t_q>t_{q1}$ (curve 1), $n_1 \leq 1$, and each query can be searched before the next enters the system. For $t_q<t_{q3}$ (curve 3), the scan time is always t_{max}, and n_3 is given by t_{max}/t_q. Curve 2 shows a solution for an intermediate value of t_q, for which $t_s<t_{max}$ and $n_2<t_{max}/t_q$. Note the relatively acute angle between curves 1 and 3. Because of this, one can expect that for most values of t_q, n=1 or $n=t_{max}/t_q$. The borderline between the system being empty and full is therefore rather narrow. However, t_{max}= 20 sec. for the 3330 disks and staggered data organization assumed, so response time is good even if all cylinders are searched on each scan.

Table 5.1 shows how n varies for differing values of p_a and t_q. The values are determined using values for 3330-technology disks (C=800, T=19, and R=.01667 sec.). Note that for $p_a=1$, $n=t_{max}/t_q$. This case corresponds to an unindexed system, in which all queries are searched for on all tracks. The column t_q=1.5 sec. closely corresponds to a system with 200 users, each generating a query every five minutes. For p_a=.001 (a reasonable value for an indexed system), $t_s<t_q$, which means a query could be completely searched before the next one arrived. At this value of p_a,

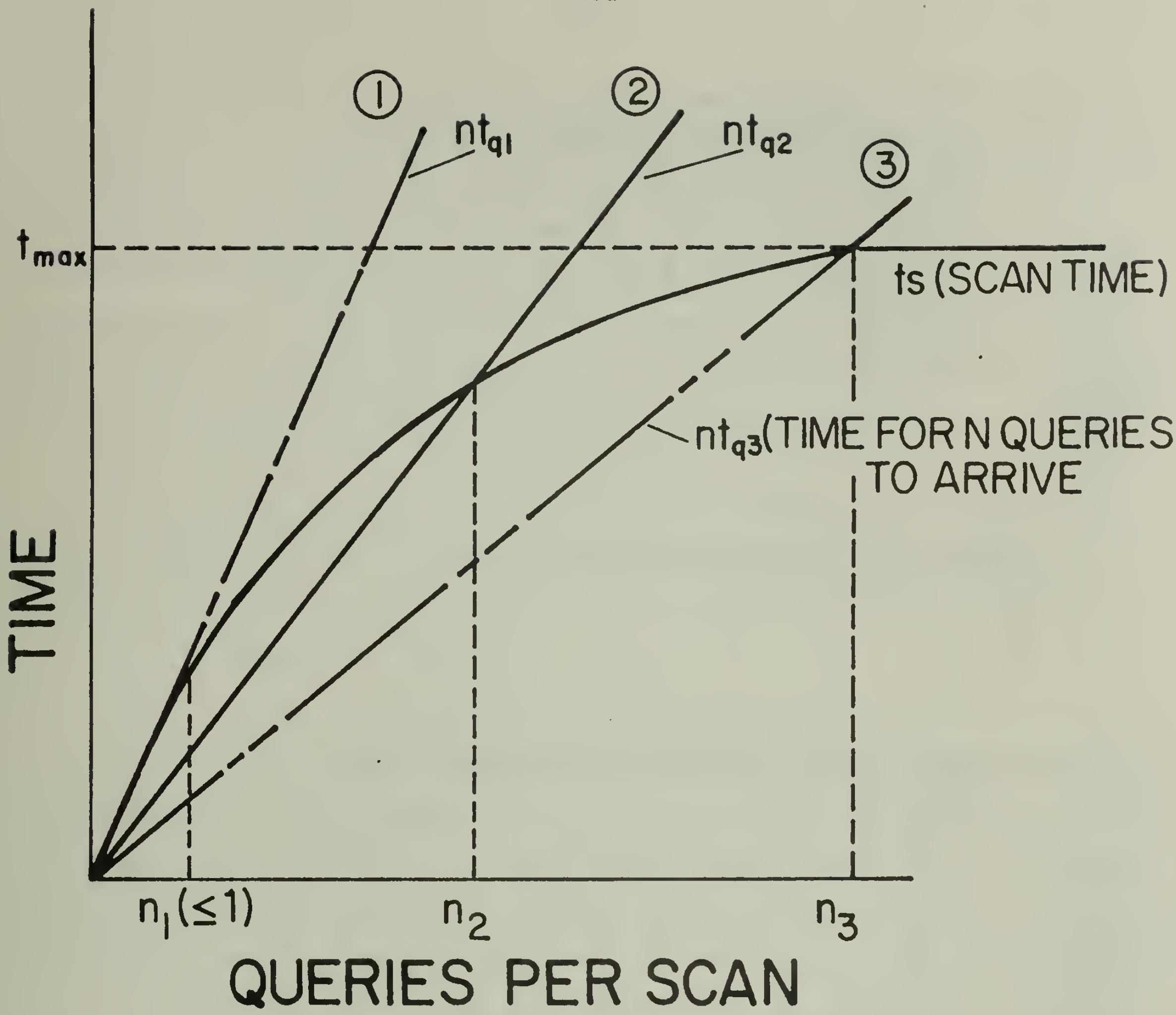


Figure 5.1 Scan Time vs. Queries per Scan

the query arrival rate could double before the system started to load up.

5.1.3 Number of Queries in System - Poisson Arrivals

The preceding section assumed that the query interarrival time t_q was constant. Since it is unlikely that users will cooperate with the system to the extent of entering queries at precisely timed intervals, the analysis must be extended to randomly distributed arrival times.

A simple, yet realistic way of handling this is to assume that queries will arrive according to a Poisson distribution with time constant t_q. The queries are then placed in a queue, and are released to the

p_a	Interarrival Time in Sec (t_q) .5	1.0	1.5	2.0	2.5	3.0	3.5	4.0	4.5	5.0
1	41	21	14	11	9	7	6	6	5	5
.3	41	21	14	11	9	7	6	6	5	5
.1	41	21	14	11	9	7	6	6	5	5
.03	41	21	14	11	8	7	6	5	5	4
.01	41	20	13	9	7	5	4	3	2	1
.003	38	14	4	1	1	1	1	1	1	1
.001	9	1	1	1	1	1	1	1	1	1
.0003	1	1	1	1	1	1	1	1	1	1
.0001	1	1	1	1	1	1	1	1	1	1
.00003	1	1	1	1	1	1	1	1	1	1

Table 5.1 - Queries Searched per Scan

p_a	Interarrival Time in Sec (t_q) .5	1.0	1.5	2.0	2.5	3.0	3.5	4.0	4.5	5.0
1	41	21	14	11	9	7	6	6	5	5
.3	20	12	9	7	6	5	5	5	4	4
.1	9	6	4	4	3	3	3	3	2	2
.03	4	3	2	2	2	2	2	1	1	1
.01	2	2	1	1	1	1	1	1	1	1
.003	1	1	1	0	0	0	0	0	0	0
.001	0	0	0	0	0	0	0	0	0	0
.0003	0	0	0	0	0	0	0	0	0	0
.0001	0	0	0	0	0	0	0	0	0	0
.00003	0	0	0	0	0	0	0	0	0	0

Table 5.2 - Queries per Searcher

p_a	Interarrival Time in Sec (t_q) .5	1.0	1.5	2.0	2.5	3.0	3.5	4.0	4.5	5.0
1	19	19	19	19	19	19	19	19	19	19
.3	19	19	19	19	19	19	19	19	19	19
.1	19	19	18	17	16	15	14	14	13	13
.03	18	14	12	10	9	8	8	7	7	6
.01	11	8	6	5	4	4	3	3	2	2
.003	6	3	2	1	1	1	1	1	1	1
.001	2	1	1	1	1	1	1	1	1	1
.0003	0	0	0	0	0	0	0	0	0	0
.0001	0	0	0	0	0	0	0	0	0	0
.00003	0	0	0	0	0	0	0	0	0	0

Table 5.3 - Searchers per Drive

system for searching at constant intervals $t_{q'}$. The analysis of t_s and n vs. t_q and p_a could then be used by substituting the release interval $t_{q'}$ corresponding to the average interarrival time t_q. This is actually a very reasonable model; before a query can be searched, various processing is necessary (building state tables, doing index processing, etc.), and performing these functions will tend to `smooth out' bursts of arrivals. Rather than the queue being artificially imposed to simplify the analysis, it is a reflection of the way the host processes queries.

The length of this queue is of interest, because if it becomes too long, response time will suffer. Since the queue is an M/G/1 system with constant service time [CofDe73], the average queue length is given by:

$$n_q = r(1-r/2)/(1-r) \quad \text{where } r=t_{q'}/t_q$$

Figure 5.2 shows how average queue length n_q varies with r. Notice that queries must be released for searching faster than the average interarrival time, or else the equilibrium queue length will be infinite. Choosing $t_{q'}=.3t_q$ results in an average queue length of .364, meaning that the queue is normally empty. However, the number of queries in the system $n'(t_{q'},p_a) \geq n(t_q,p_a)$ for $t_{q'}<t_q$ (see Figure 5.1 and Table 5.1). Thus, choosing too small a value of $t_{q'}$ (i.e. releasing queries to the system at a faster rate) can result in a larger number of queries in the system. This increases the scan time t_s, which hurts response time. The value of $t_{q'}$ must be chosen to balance degraded response due to time spent in the queue against increased response time <u>and</u> system load due to many queries being searched. For example, choosing an intermediate value of $t_{q'}=.8$ results in a queue length of only 2.4, without significantly effecting scan time.

5.1.4 Queries per Searcher

If several queries are being searched, it is possible that more than one will require searching the same track. This will certainly be the case in a nonindexed system, where all queries must be searched for on all tracks. The probability that a given query accesses a given track (p_a) was assumed to be uniform for all tracks. This is reasonable if documents

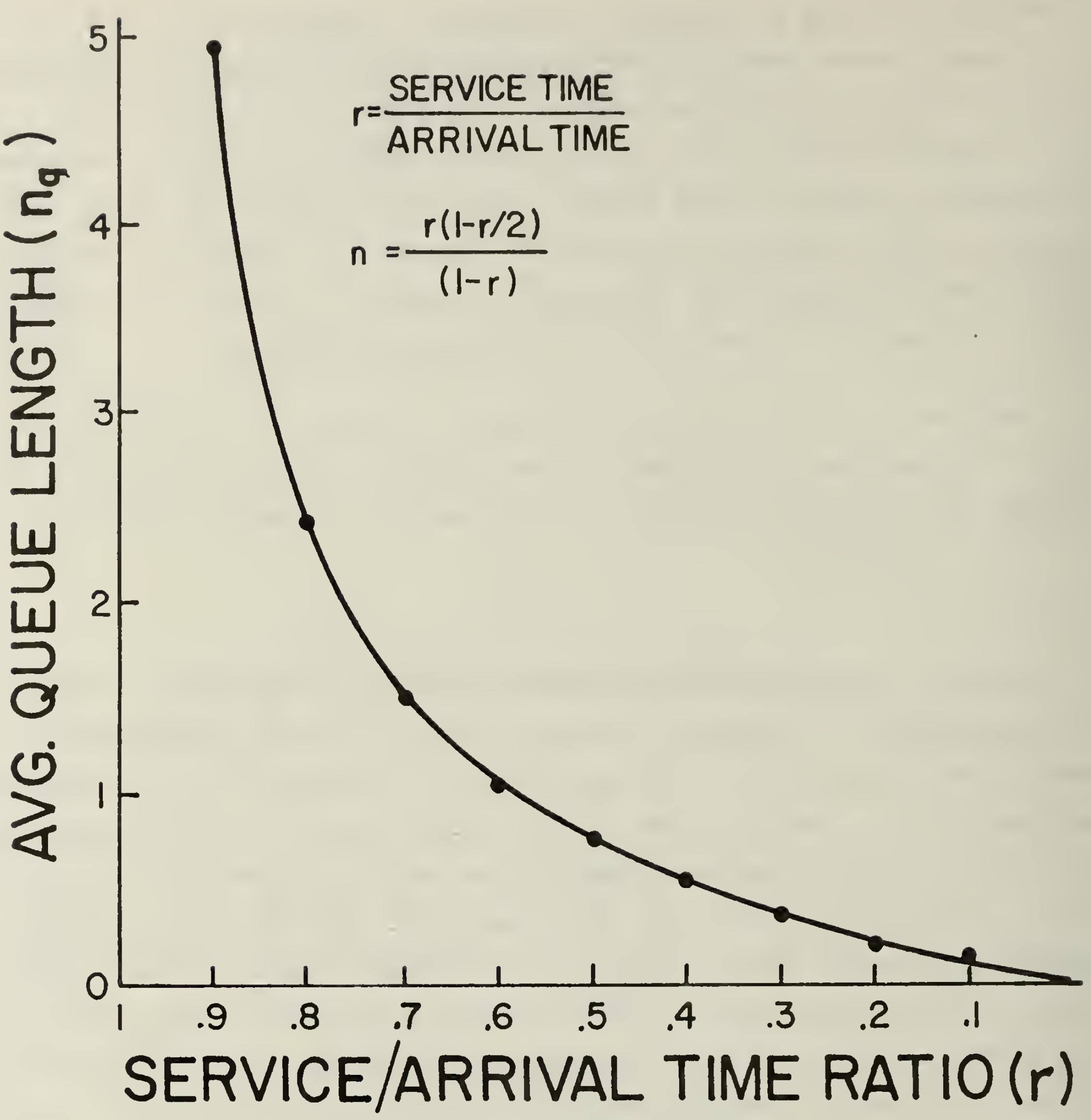


Figure 5.2 Average Queue Length for Poisson Arrivals

are laid out on the disk at random, and if relatively few search contexts span tracks. Since one searcher scans each track, it must be capable of containing all queries mapping onto the track. Under these assumptions, the probability that m or fewer queries access a given track is:

$$\Pr[\#\text{ queries} \leq m] = \Sigma_{k=0}^{m} \frac{n!}{k!(n-k)!} p_a^{\,k} (1-p_a)^{n-k}$$

where n is the number of queries in the system, calculated as shown in Section 5.1.1.

The number of queries accessing a track is important because it affects the design of the hardware text searcher. The more queries that access a track, the more terms there are to be searched for. This means that more CM's will be required and that the term hit rate will be higher. If the CM's cannot contain the terms for all queries, it will be necessary to search the track in multiple revolutions.

The term matcher should be configured to be able to handle a sufficiently large batch of terms that only a small percentage of cylinders require more than one revolution to search. Table 5.2 shows m (the number of queries per searcher) such that Pr[# queries mapping onto a track $\leq$ m] $\geq$ .99. As one would expect, for p_a=1, each searcher must look for all queries in the system (see Table 5.1). However, for a partially indexed system with p_a=.001 and t_q=1.5 (the example used above), over 99% of the time no queries will be mapped onto a track, and almost never will more than one. Thus, a searcher designed to handle only one query will be sufficient.

5.1.5 Searchers per Drive

If cylinders are to be searched in one revolution, sufficient searchers must be built into each disk drive to handle each track requiring searching. If p_t is the probability that a track requires searching, the probability of needing to search k or fewer tracks on a cylinder is given by:

$$\Pr[\#\text{ tracks} \leq k] = \Sigma_{j=0}^{k} \frac{T!}{j!(T-j)!} p_t^{\,j} (1-p_t)^{T-j}$$

As in the last section, to configure a drive with the proper number of searchers, one must decide on a threshold probability that enough searchers will be available, and choose k accordingly. Table 5.3 shows values for k for varying p_a and t_q, again assuming Pr[# tracks being searched on cylinder $\leq$ k]$\geq$.99. Not surprisingly, a nonindexed system

(p_a=1) requires 19 searchers, one for each track of the cylinder. Notice that for the example partially indexed system (p_a=.001, t_q=1.5 sec.), the probability is over 99% that no more than one track on a cylinder will require searching. For this system, drives may safely be configured with only one searcher. As the previous section showed, this searcher only has to accommodate one query. Chapters 2 and 4 showed that the searcher could be implemented using very few IC packages, so in this case the hardware searcher would increase the cost of the disk drive by only a small fraction.

5.2 Term Matcher Characteristics

Section 5.1.4 showed how the maximum likely number of queries per searcher could be determined from the interarrival time (t_q) and the index system efficency (p_a). If statistics regarding the number of terms per query are available, the number of terms that the matcher should be configured to handle can be determined.

For most indexed systems, the searcher will be designed to handle only one query. Evidence is confusing regarding just how large one query should be expected to be. [OSI77a] gave estimates for the CIA's SAFE system, indicating that the average number of terms per query was 23. However, the references for this system are not consistent about this. Roberts ([Rob77]) gave a conflicting set of requirements indicating that the average number of terms per query would be 41.2, and that 10% would have an average of 70 terms. How these estimates were obtained was not stated. Milner ([Miln76]) gave statistics from an analysis of one day's transactions of MEDLINE. His results indicated that the average number of terms per query was 6.5, even using the EXPLODE (thesaurus) feature. However, about 1/8 of the terms used in queries caused explodes. The average number of basic terms per exploded term in MEDLINE was given as 30.25.

Again, there are fundamental differences between these systems and a system using the search hardware that has been discussed in previous chapters. SAFE uses one Bird FSA per drive, and the average response time is 7.5 minutes, which hardly encourages interactive use of the system.

Intuitively, users would try to minimize the number of searches they perform, which could explain the relatively large average number of terms per query. Additionally, the SAFE system is designed to be used in a rather specialized manner. The database is composed of continuously updated intelligence data, and the users are primarily specialists in a given area looking for new information pertinent to their specialty. Such users, looking for the same type of information day after day, have time to refine their queries. Rather than performing their intellectual search as a sequence of smaller trial queries, they can input the same (larger) canned query each time and have a fair amount of confidence that it will produce the correct result. These factors would seem to contribute to the large expected number of terms per query.

Substantial differences also exist between MEDLINE and a true full-text retrieval system; MEDLINE only stores document citations and only indexes on a small subset of the terms in the document. However, MEDLINE has a relatively fast response time (computed as averaging 11.83 sec. in [Miln76]), and therefore might have query characteristics similar to those of an interactive full-text system. Since a significant fraction of the queries contain exploded terms, the matcher should be able to accommodate in the vicinity of 30-50 terms.

Once the number of terms to be handled by the term matcher is decided upon, the number of CM's needed and the number of states per CM can be determined. For tables of under 120 terms, Figure 3.8 can be used to determine the number of CM's. To handle the 30-50 terms in the MEDLINE example, seven CM's would be required. Figure 3.5 showed that the average number of states per term was very close to seven, so for 50 terms, the table would have around 350 states. The number of states per CM would be (350 states/7 CM's) = 50. Rounding up to the next power of 2, each CM would need 64 states. Thus, to handle a large query, the term matcher would be configured with 7 CM's with 64 states each. In cases where the searcher has to handle more than one query, this configuration could accommodate 7.6 `average' 6.5-term queries.

As another example, requirements for the SAFE system given in [OSI77a] specify a maximum of 70 active queries per scan, which corresponds to an interarrival time t_q=3.5 sec. Table 5.2 shows that this corresponds to 6 queries per searcher. Since the same reference gives the number of terms per query as 23.57, each term matcher must handle 142 terms. Extrapolating Figure 3.8 slightly, this number of terms requires 13 CM's. Each CM would require 76 states. Rounding up to the next power of two, the configuration for this system would use a matcher containing 13 CM's with 128 states each.

If the NFSA Term Matcher was built using LSI, the design and layout would be the most important factor in its cost. It is not likely that designing CM's with different numbers of states would be practical. Therefore, one would like to choose a memory size able to accommodate any system's requirement. Figure 3.8 illustrated that the number of CM's required was 5+(t/16), where t is the number of terms. Thus, the number of terms per CM is asymptotic to 16. Since each term contains about seven states, each CM need contain only 112 states. Again, the next highest power of two is 128. Therefore, assuming that the terms/CM ratio continues to grow linearly, CM's of 128 states will be adequate for tables of arbitrary size.

5.3 Loading the Searchers

The searchers will be loaded while the disk is seeking to the next cylinder. During this interval, it is necessary to load each searcher on a drive with the search tree (residing in the query resolver), and the startup and state tables for each CM in the searcher. Each search tree node requires 40 bits, and since leaves are not represented by nodes, only about one node per term is necessary. Each state table entry requires an average of 18 bits (16 for the TT word, plus a 1/8 probability of needing a 10-bit fork table entry). Additionally, the startup table for each CM contains (64x13)=832 bits. Assuming that t terms of seven states each are loaded into n CM's, the number of bits to load into each searcher is:

$$(\text{bits/searcher}) = 832n + t(7\times 18 + 40)$$

Substituting the formula n=(t/16)+5 derived from Figure 3.8 into the above equation,

$$(\text{bits/searcher}) = 4160 + 832t/16 + 166t$$

$$= 4160 + 218t$$

The time available to load all searchers on a drive is 1/2 revolution, or 8 milliseconds. Assuming a data rate of 10^7 bits/second,

$$t = [(8\text{x}10^4/m) - 4160] / 218$$

where m is the number of searchers being loaded (i.e. the number of tracks being searched for on the cylinder). Figure 5.3 shows how k varies with

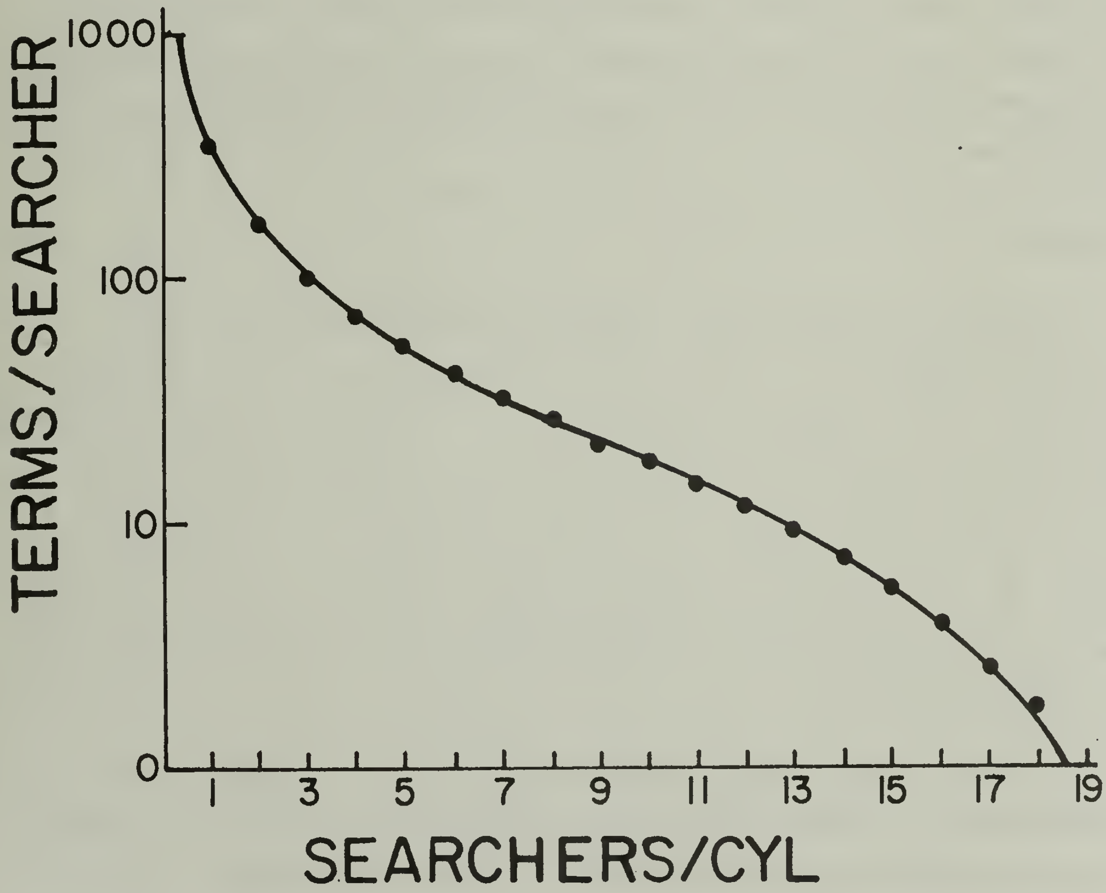


Figure 5.3 Searcher Capacity as Limited by Loading Time

m. Note that if only one searcher is being loaded, there is time to load it with 349 terms. There is more than adequate bandwidth to handle the indexed system example (p_a=.001, t_q=1.5 sec.).

It initially seems alarming that if all 19 tracks require searching, there is not even time to load one term. However, in practice, any system using an index will have a very low value of p_a ($\ll$.01), and thus will search at most a few tracks per cylinder (Table 5.3). Only nonindexed systems would be configured with a searcher per read head. In a nonindexed system, all searchers will be looking for the entire query batch, and will therefore be loaded with identical tables. If the control logic in the drive is appropriately designed, it is only necessary to send the tables once, broadcasting them to all searchers. Actually, the situation is even better than this; the tables need be reloaded at most each time a query enters or leaves the system, which is considerably less often than once per disk revolution. In fact, by batching queries and processing them together, the searchers would only have to be loaded once per 20-second scan.

5.4 A Case Study

To demonstrate how a NFSA searcher might be configured for a practical application, we will again consider the example of the SAFE system mentioned briefly in Section 5.2. To reiterate, [OSI77a] estimated the minimum query interarrival time t_q as 3.5 seconds and the average number of terms per query as 23. Naturally, since the system is not indexed, all queries search all tracks and thus p_a=1. The statistics shown in Figure 3.5 suggest that each query will contain around 177 characters (7.7 characters per term) and will require around 168 states (7.3 states per term).

To evaluate the practicality of using the NFSA searcher, the amount of hardware necessary will be compared against that for the Bird FSA used by the SAFE system. Most of the gates in both architectures are in the state table memories, so the state table memory size will be compared for the two designs. For a given table size, two numbers will be presented: the total number of bits in the state table memory (which is a good

measure of system cost, but due to packaging considerations can be larger than what is actually needed), and the number of bits used to store the benchmark state table (which is a measure of load time and indicates the room available for larger than average tables).

The SAFE system was designed to use one Bird FSA per disk drive. The entire disk is scanned repeatedly to search, which takes about 266 seconds. Queries are collected during each scan for searching during the next scan. The average response time is therefore about 1.5 scans, or 400 seconds (6.67 minutes). Configuring a comparable system with NFSA searchers would require 19 searchers per disk drive (one per read head), and using the staggered data organization assumed throughout this chapter, would require only 20 seconds to scan the entire disk surface.

Two variants of the Bird FSA were mentioned in the SAFE design, one processing 6-bit characters, and the other processing each data byte from the disk as two sequential 4-bit nibbles. Each has different memory requirements, but using figures from [OSI77a], a six-bit character version configured to be capable of handling the assumed 70 queries per scan would require a total of 1392640 bits of memory (4 boards of 4096 85-bit words each), of which 678640 would actually be used. The 4-bit nibble version would require 5 boards of 4096 39-bit words, or 798720 bits of memory. Of these, 622758 would actually be used. In comparison, the NFSA matcher would be configured with 13 CM's of 128 states (including 16 fork states) each. Each CM would require (64x13)=832 bits of ST memory, (128x16)=2048 bits of TT memory, and (16x10)=160 bits of FT memory. The 19 NFSA searchers per drive would thus require a total of 750880 bits, of which 550240 would actually be used. The amount of memory required by the NFSA searcher compares quite favorably with that required by either version of the Bird FSA, <u>even ignoring</u> that the latter requires 100 nanosecond memory and the former only requires a memory speed in the vicinity of 600 nanoseconds. When it is remembered that response time is at least 13 times better for the NFSA searcher, this design seems even more attractive.

Admittedly, the use of an index might not be practical considering the SAFE system's constantly changing database of intelligence data. However, it is interesting to look at how the system could be configured if indexing were used. Assuming Poisson arrivals with an average interarrival time t_q=3.5 seconds, releasing queries into the system every $t_{q'}$=2.5 seconds results in an average queue length n_q=1.5 (Figure 5.2, r=.7). If an index is used with p_a=.001, column 5 of Table 5.2 shows that each searcher would only have to be designed to search for one query. The average query is assumed to contain 23 terms, and configuring each matcher with 6 CM's would be sufficient to handle queries of up to 32 terms. Each CM requires (13x64)=832 bits for the ST memory, and assuming that each CM has 128 states (16 fork states), each would have 2304 bits of state table memory. Each matcher would thus have 6x(832+2304)=18816 bits of memory. The 168 states in the average query require (168x18)=3204 bits, so of the 18816 total bits in the matcher, only (832x6)+3204, or 8196 bits, would actually be used. Since Table 5.3 shows that only one searcher per drive is needed, this is the total system memory requirement. Finally, as Table 5.1 shows, each query can be searched in (much) less than the release time $t_{q'}$. Neglecting index processing time, the total response time is the search time plus the time spent in the queue, which is $2.5t_{q'}$=6.25 seconds.

As a final example, consider a situation in which indexing is not used, and off-the-shelf 3330-type disks are used having only the capability of reading from one track at a time. This is identical to the SAFE system implementation. In this case, 70 queries (1610 terms) would have to be searched for at once. The memory requirements were given above for two variations of the Bird FSA, but suppose that an NFSA searcher was used instead. If one large NFSA were used, a table of 11753 states would have to be assigned to a term matcher with 95 CM's (extrapolating the results of Chapter 3). This is not practical, so instead (since the 70 queries can be processed independently), several term matchers could be connected in parallel to the one read head, either reporting hits to one query resolver or to several (depending upon hit rate). If each NFSA term matcher was configured to have 16 CM's of 128 states each, seven 23-term queries would comfortably fit in each. Thus, a total of 10 term matchers

can contain the 70 queries. Each CM has 3040 bits of memory, so the 160 CM's necessary to handle the 70 queries would have a total of 486400 bits of 600-nanosecond memory, of which 348034 would actually be used. This is slightly more than 60% of the 798720 bits of 100-nanosecond memory necessary for the 4-bit nibble processor version of the Bird FSA.

5.5 Summary

This chapter considered how the NFSA searcher could be configured for different types of systems. The two parameters having the most impact on how the system is configured are the query interarrival time and the index system efficiency (which indicates how much data must be searched for each query). A simple model was developed showing how search time, response time, searcher size, and the number of searchers per drive vary for a wide spectrum of potential systems with different values of the two parameters. It was shown that for two representative systems with very different parameters (indexed and nonindexed), systems could be configured having excellent response time and (for the indexed system) almost negligible cost for the search hardware. Finally, the performance and cost (in terms of memory requirements) of the Bird FSA searcher and the NFSA searcher were compared. The NFSA searcher can be implemented in LSI, which allows duplicating it so that each read head can be connected to its own searcher. This allows the response time to be 13 times faster than that for the SAFE system using the Bird FSA. Additionally, the NFSA version can use much slower memory, and needs less of it than the Bird FSA. Finally, even if the NFSA is used in the same manner as the Bird FSA (i.e. connected to one head of the disk, searching the entire surface sequentially), the memory requirement is still much lower. As Chapter 1 mentioned, the Bird FSA seemed to be the most attractive of the systems mentioned in previous literature. The NFSA searcher seems to be a significant improvement over the Bird FSA, and all evidence seems to support the contention that systems can be built economically to allow many users to simultaneously search very large text databases with excellent response time.

Chapter 6

Summary and Conclusions

Very large text retrieval systems could prove very useful in many areas such as law, medicine, science, and engineering. The main obstacle to building such systems is the inability of conventional, software based search techniques to deliver satisfactory response times at reasonable cost. To get fast response time, the database must be searched in parallel, and to do this cost-effectively requires special purpose hardware optimized for the task of text searching.

The major components of a large-scale text-retrieval system are the host computer, the index processor, and the text searcher. This thesis focused on the latter. It was shown that previous architectures for hardware text searchers had problems inhibiting their use in large systems. A new architecture was introduced, which is modeled after a nondeterministic finite-state automaton (NFSA). It was discussed how this searcher could be used to search not only for words, but also for many other interesting classes of patterns.

It was shown how the NFSA could be implemented in hardware using interconnected sub-machines. The state table is partitioned among the sub-machines, insuring that each will only be following one alternative search path at once. The particular organization chosen allowed a simple comparator to be used in each machine (CM). It avoided state table memory contention by storing each CM′s states locally, and included self-starting and forking features in each CM to eliminate the necessity of externally scheduling the CM′s.

Although partitioning the state table into compatible subsets facilitated the hardware implementation of the searcher, it was shown that a simpleminded approach to computing the partitions was too slow to be used in production. Optimizations were developed which drastically decreased the amount of computation necessary compared to that required by

direct application of standard table partitioning algorithms. It was shown that state tables of reasonable size could be partitioned in real time even on a relatively slow machine.

The NFSA Matcher is capable of recognizing terms and other primitive patterns in the database, but to detect instances of higher-level search expressions generated by user queries, separate query-resolution logic is necessary. The fact that the NFSA Matcher runs at disk speeds places severe speed constraints on the query resolver, and methods of doing resolution under these constraints were discussed. Methods were also discussed for resolving hits in contexts spanning physical device boundaries.

For it to be practical to actually build text searching hardware, it must be applicable to systems with possibly widely varying characteristics. A major advantage of the NFSA Matcher is that it is made up of small, identical building blocks (Character Matchers). As such, it can be configured to efficiently handle a wide range of anticipated system loads and indexing strategies. It was studied how these parameters influence the amount of hardware necessary, and how the NFSA Matcher could be configured for an individual system once values for these parameters are known. Examples were presented to show that the amount of hardware necessary for a practical system was indeed reasonable, and was in fact less than that required by the best previous searcher architecture.

Although the NFSA Searcher was discussed from the viewpoint of its application to searching very large database, there is no basic reason why it cannot be used in other environments. Its high bandwidth was achieved by breaking the database down into many independent data streams and searching them simultaneously. Since the hardware necessary to search each data stream is relatively inexpensive (especially if only one query is being searched for), the NFSA Searcher 'scales down' well to smaller systems. For example, in an office automation system, a small NFSA Searcher could be used in the local terminals to provide a fast-access 'automated filing cabinet' feature.

The NFSA might also be useful to search text stored in the memory of a mainframe CPU. Although the search bandwidth for an individual NFSA is lower than one might hope for in such an application, several obvious methods exist for speeding it up. First of all, remember that the match rate is governed by the logic family used to implement the searcher, and for the LSI version which searched data directly from disk, very slow logic and memory were adequate. If the NFSA were used to augment the searching capability of a mainframe, not as many duplicates of the NFSA would be necessary, and the economic factors that forced an LSI implementation to be used for the database application would no longer hold. Therefore, faster logic families could be used. Memories capable of storing the state table exist with access times as low as 20 nanoseconds, and correspondingly fast logic is available for implementing the comparitors and registers. While no detailed timing estimates for such a matcher were made, it seems possible to achieve match rates of under 50 nanoseconds per character. Depending upon the application, it might also be possible to use the search strategy that was used for databases, that is, breaking down long strings to be searched into smaller segments and searching these in parallel with slower, cheaper searchers. If the patterns being searched for are short relative to the string segment size, many searchers could be used in this manner to match segments of long strings at a very high aggregate bandwidth.

As with any project of this sort, many extensions and possibilities for future research were thought of that could not be pursued due to time limitations. First, several extensions to the basic NFSA design are possible to greatly extend its power. One possibility is including the capability to do matches on numbers stored in the database. Remember that the comparator in the MSI prototype discussed in Chapter 2 had the capability of doing full numeric comparisons (<, =, and >). If numbers to be searched (e.g. dates) are stored using a fixed number of digits with leading zeroes, a simple extension to the NFSA could allow a state table to be defined to search for numbers related to a pattern by one of the above three operators. Another extension might be to support searching for `near misses' on a pattern, allowing a limited number of incorrect characters to occur before a search path is abandoned. This would be

useful in a non-indexed system (such as an office automation system) where spelling errors are possible.

But perhaps the major area needing further work is investigating how to tie all the parts of a large database system together. As Chapter 1 mentioned, index processing has been studied in some depth, and now similar work has been done on text searching. Integrating index processing hardware and text searching hardware into a balanced system will require much thought. Partial inversion allows impressive speedup of the search using a relatively small index, while still allowing the full flexibility in search operations possible with full-text searching. However, the level to which the text is inverted must be properly chosen if the workload is to be balanced between the index processor and the searcher. Additionally, data flow between the components of the system must be studied. The output of the index processor is a set of postings not necessarily ordered by disk address. These must be sorted to allow commands to the text searchers to be generated. These and many other problems dealing with the care and feeding of the index processing and text searching hardware require further study.

The light is at the end of the tunnel - the parts of a large hardware-augmented search system are now all understood, and there is some feeling for how they can be fit together. With just a little more work on the higher-level system aspects, the use of specialized processors to search very large text databases in interactive time will become a reality.

References

[Bayer78] Bayer, M. P., "Dialog - An Online Retrieval System for Bibliographic Information," Digest of Papers, COMPCON Fall 1978, Washington, D. C., pp. 54-58.

[BiNeTr78] Bird, R. M., Newsbaum, J. B., and Trefftzs, J. L., "Text File Inversion: An Evaluation," Proc. Fourth Non-Numeric Workshop, Syracuse, N. Y., Aug. 1978, pp. 42-50.

[BiTuWo77] Bird, R. M., Tu, J. C., and Worthy, R. M., "Associative/Parallel Processors for Searching Very Large Textual Data Bases," Proc. Third Non-Numeric Workshop, Syracuse, N. Y., May, 1977, pp. 8-16.

[Black78] Black, D. V., "System Development Corporation's Search Service," Digest of Papers, COMPCON Fall 1978, Washington, D. C., pp. 59-64.

[Booth68] Booth, T. L., Sequential Machine and Automata Theory, John Wiley and Sons, Inc., New York, 1968.

[BurEm79] Burket, T. G., and Emrath, P. E., "User's Guide to Eureka and Eurup," Report No. UIUCDCS-R-79-956, University of Illinois, Urbana, Feb. 1979, pp. 1-49.

[ChKu77] Chen, S. C., and Kuck, D. J., "Combinational Circuit Synthesis with Time and Component Bounds," IEEE Transactions on Computers, Vol. C-26, No. 8, Aug 1977, pp. 712-726.

[Cheng77] Cheng, W. K., "Multiprocessor for String Manipulation," M. S. Thesis, University of Illinois, Urbana, Oct. 1977, pp. 22-65.

[CofDe73] Coffman, E. G., and Denning, P. J., Operating Systems Theory, Prentice-Hall, Englewood Cliffs, New Jersey, 1973.

[Cop78] Copeland, G. P., "String Storage and Searching for Data Base Applications: Implementation on the INDY Backend Kernel," Proc. Fourth Non-Numeric Workshop, Syracuse, N. Y., Aug. 1978, pp. 8-17.

[FoKu80] Foster, M. J., and Kung, H. T., "Design of Special Purpose VLSI Chips: Examples and Opinions," Carnegie-Mellon University, Pittsburgh, Pa., Sept. 1979, pp. 5-17.

[FrMe75] Friedman, A. D., and Menon, P. R., Theory and Design of Switching Circuits, Computer Science Press, Woodland Hills, California, 1975.

[HaSt66] Hartmanis, J., and Stearns, R. E., Algebraic Structure Theory of Sequential Machines, Prentice-Hall, Englewood Cliffs, New Jersey, 1966.

[Hollaar76] Hollaar, L. A., "An Architecture for the Efficient Combining of Linearly Ordered Lists," Second Workshop on Comp. Arch. for Non-

Numeric Processing, Jan. 1976.

[Hollaar79] Hollaar, L. A., "Text Retrieval Computers," Computer, March 1979, Vol. 12, No. 3 (ISSN 0018-9162), pp. 40-50.

[HopU169] Hopcroft, J. E., and Ullman, J. D., Formal Languages and Their Relation to Automata, Addison-Wesley, Reading, Massachusetts, 1969.

[HopU179] Hopcroft, J. E., and Ullman, J. D., Introduction to Automata Theory, Languages, and Computation, Addison-Wesley, Reading, Massachusetts, 1979.

[HsKaKe75] Hsiao, D. K., Kannan, K., and Kerr, D. S., "Structure Memory Designs for a Database Computer," Proc. ACM Annual Conf., 1977, pp. 343-350.

[Hurl76] Hurley, B. J., "Analysis of Computer Architectures for Information Retrieval," M. S. Thesis, University of Illinois, Urbana, May 1976.

[LiSmSm76] Lin, C. S., Smith, D. C. P., and Smith, J. M., "The Design of a Rotating Associative Memory for Relational Database Applications," ACM Trans. Database Syst., March 1976, Vol. 1, No. 1, pp. 53-65.

[McCar78] McCarn, D. B., "Online Services of the National Library of Medicine," Digest of Papers, COMPCON Fall 1978, Washington, D. C., pp. 48-53

[MeCon80] Mead, C., and Conway, L., Introduction to VLSI Systems, Addison-Wesley, Reading, Massachusetts, 1980.

[Miln76] Milner, J. M., "An Analysis of Rotational Storage Access Scheduling in a Multiprogrammed Information Retrieval System," Ph. D. Thesis, University of Illinois, Urbana, Sept. 1976.

[Morgan76] Morgan, J. K., "Description of an Experimental On-Line Minicomputer-Based Information Retrieval System," M. S. Thesis, University of Illinois, Urbana, Feb. 1976.

[Muk78] Mukhopadhyay, A., "Hardware Algorithms for Non-numeric Computation," Proc. Fifth Symposium on Computer Architecture, Palo Alto, Calif., Apr. 1978, pp. 8-16.

[MuWar79] Mules, D. W., and Warter, P. J., "A String Matcher for an 'Electronic File Cabinet' Which Allows Errors and Other Approximate Matches," Department of Electrical Engineering, University of Delaware, Newark, April, 1979.

[OzScSm75] Ozkarahan, E. A., Schuster, S. A., and Smith, K. C., "RAP: an Associative processor for Data Base Management," Proc. 1975 AFIPS Nat. Comp. Conf., Vol. 44, AFIPS Press, Montvale, N. J., pp. 379-387.

[OSI77a] "High-Speed-Text-Search Design Contract Interim Report," OSI:R77-002, Operating Systems Incorporated, Woodland Hills, Calif., Jan. 1977, pp. 2-69 - 2-101.

[OSI77b] "High-Speed-Text-Search Design Contract Design Specification Document," OSI:R77-008, Operating Systems Incorporated, Woodland Hills, California, April 1977.

[Rob77] Roberts, D. C. (ed.), "A Computer System for Text Retrieval: Design Concept Development," U. S. Central Intelligence Agency ORD, RD-77-10011, Oct. 1977.

[SinSh72] Sinha Roy, P. K., and Sheng, C. L., "A Decomposition Method of Determining Maximum Compatibles," IEEE Transactions on Computers, Vol. C-21, March 1972, pp. 309-312.

[Sprowl76] Sprowl, J. A., "Computer-Assisted Legal Research - An Analysis of Full Text Document Retrieval Systems, Particularly the LEXIS System," American Bar Foundation Research Journal, Jan. 1976, Vol. 1, No. 1, pp. 175-226.

[Stell74a] Stellhorn, W. H., "An Experimental Information Retrieval System," Report No. UIUCDCS-R-74-657, University of Illinois, Urbana, July 1974, pp. 22-43.

[Stell74b] Stellhorn, W. H., "A Processor for Direct Scanning of Text," presented at First Nonnumeric Workshop, Dallas, Oct. 1974.

[Stell75] Stellhorn, W. H., "A Specialized Computer for Information Retrieval," Ph.D. Thesis, University of Illinois, Urbana, 1975.

[Stoff74] Stoffers, K. L., "Sequential Algorithm for the Determination of Maximum Compatibles," IEEE Transactions on Computers, Vol. C-23, Jan. 1974, pp. 95-98.

[SuLip75] Su, S., and Lipovski, G. J., "CASSM: A Cellular System for Very Large Databases," Proc. Conf. on Very Large Data Bases, Sept. 1975, pp. 456-472.

[Unger69] Unger, S. H., Asynchronous Sequential Switching Circuits, Wiley-Interscience, New York, 1969, pp. 28-63.

[Woll80] Wollsen, D. L., "CMOS LSI - The Computer Component Process of the 80's," Computer, ISSN 0018-9162, February, 1980, pp. 59-67.

Vita

Roger Lee Haskin was born on August 8, 1950, in Cleveland, Ohio. He received his B.S. degree in Computer Engineering from the University of Illinois at Urbana-Champaign in February, 1973, and his M.S. in Computer Science from the same institution in October, 1978.

Mr. Haskin was a senior systems engineer with Datalogics, Inc., in Chicago, Illinois from 1972 to 1975. He supervised a group developing a comprehensive computer typesetting system which supported multicolumn full-page makeup. Since 1975, he has been a research assistant at the University of Illinois at Urbana-Champaign, where he has done work in the fields of artificial intelligence, computerized aircraft flight instrumentation, information retrieval, and computer architecture.

BIBLIOGRAPHIC DATA SHEET	1. Report No. UIUCDCS-R-80-1027	2.	3. Recipient's Accession No.
4. Title and Subtitle HARDWARE FOR SEARCHING VERY LARGE TEXT DATABASES			5. Report Date August 1980
			6.
7. Author(s) Roger Lee Haskin			8. Performing Organization Rept. No. UIUCDCS-R-80-1027
9. Performing Organization Name and Address University of Illinois at Urbana-Champaign Department of Computer Science Urbana, IL 61801			10. Project/Task/Work Unit No.
			11. Contract/Grant No.
12. Sponsoring Organization Name and Address National Science Foundation Washington, D.C.			13. Type of Report & Period Covered Doctoral-1980
			14.

15. Supplementary Notes

16. Abstracts

The problem of searching very large text databases is discussed. Problems with using both conventional CPU's and previously developed text search hardware are noted. A new model for text searching; using a nondeterministic finite-state automaton (NFSA) to control matching, is introduced. By partitioning the nondeterministic state table and assigning blocks of states to interconnected sub-machines, it is shown how the NFSA searcher can be built with simple hardware amenable to LSI implementation. Methods of effectively partitioning large state tables are developed, query resolution is discussed, and system configuration and performance as a function of user local and other parameters is discussed.

17. Key Words and Document Analysis. 17a. Descriptors

Computer Architecture
Disk Systems
Finite-State Automata
Full-Text Document Retrieval
Inverted File
Large-Scale Integration
On-Line Information Retrieval

17b. Identifiers/Open-Ended Terms

17c. COSATI Field/Group

18. Availability Statement	19. Security Class (This Report) UNCLASSIFIED	21. No. of Pages 155
Release Unlimited	20. Security Class (This Page UNCLASSIFIED	22. Price

FORM NTIS-35 (10-70) USCOMM-DC 40329-P71